Efficiency Improvements of Iterative Numerical Algorithms on Modern Architectures

Verbesserung der Effizienz für iterative numerische Algorithmen auf modernen Architekturen

Der Technischen Fakultät der Universität Erlangen-Nürnberg zur Erlangung des Grades DOKTOR–INGENIEUR

vorgelegt von

Dipl.-Ing. Jan Treibig

Erlangen — 2008
Als Dissertation genehmigt von

der Technischen Fakultät der

Universität Erlangen-Nürnberg

Tag der Einreichung: 22.12.2008
Tag der Promotion: 18.6.2009
Dekan: Prof. Dr.-Ing. habil. J. Huber
Berichterstatter: Prof. Dr. U. Rüde
                Prof. Dr. M. Philippsen
Abstract

For many numerical codes the transport of data from main memory to the registers is commonly considered to be the main limiting factor to achieve high performance on present micro architectures. This fact is referred to as the memory wall. A lot of research is targeting this point on different levels. This covers for example code transformations and architecture aware data structures to achieve an optimal usage of the memory hierarchy found in all present micro architectures. This work shows that on modern micro architectures it is necessary to also take the requirements of the Single Instruction Multiple Data (SIMD) programming paradigm and data prefetching into account to reach high efficiencies.

In this thesis the chain from high level algorithmic optimizations over the code generation process involving the compiler and the limitations and influences of the instruction set architecture down to the micro architecture of the underlying hardware are analyzed. As a result we present a strategy to achieve a high efficiency for memory bandwidth limited algorithms on modern architectures. The success of this strategy is shown on the algorithmic class of grid based numerical linear equation solvers: A 2D Red-Black Gauss-Seidel smoother implementation for the x86/x86-64 architecture and a 3D multigrid implementation for the IA64 architecture.
I would like to thank my advisor Prof. Dr. Ulrich Rüde for supporting me in my effort to finish this thesis. Thank you for your patience and willingness to discuss all kinds of topics with real interest and great analytic expertise. I also wish to thank Prof. Dr. Michael Philippsen for reviewing this thesis.
Special thanks to all of my current and former friends and colleagues at the Lehrstuhl für Systemsimulation. Thank you for the discussions and for your patience for bearing my sometimes rude speeches.
I also wish to thank the students with whom I had the pleasure to collaborate with: Simon Hausmann, Markus Stürmer and Silke Bergler. Special thanks to Markus Stürmer who contributed many important pieces to my research.
I also wish to thank Dr. Gerhard Wellein and Dr. Georg Hager from the Regional Computing Center in Erlangen for their support and technical advisory.
Last but not least I wish to thank my girl friend Niggi and my parents for their believe in me.

Jan Treibig
ACKNOWLEDGMENTS
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>List of Figures</strong></td>
<td>xii</td>
</tr>
<tr>
<td><strong>List of Tables</strong></td>
<td>xiv</td>
</tr>
<tr>
<td><strong>List of Listings</strong></td>
<td>xvi</td>
</tr>
<tr>
<td><strong>Acronyms</strong></td>
<td>xvii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Scope</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Developments in Computer Architecture</td>
<td>3</td>
</tr>
<tr>
<td>1.4 The Hardware Software Interface</td>
<td>5</td>
</tr>
<tr>
<td>1.5 The Optimization Process</td>
<td>5</td>
</tr>
<tr>
<td>1.6 Outline</td>
<td>6</td>
</tr>
<tr>
<td><strong>I Background</strong></td>
<td>7</td>
</tr>
<tr>
<td><strong>2 Computer Architecture: Understanding the machine</strong></td>
<td>9</td>
</tr>
<tr>
<td>2.1 Instruction Level Parallelism</td>
<td>9</td>
</tr>
<tr>
<td>2.1.1 Instruction Pipelining</td>
<td>10</td>
</tr>
<tr>
<td>2.1.1.1 How to prevent data hazards</td>
<td>12</td>
</tr>
<tr>
<td>2.1.1.2 How to prevent control hazards</td>
<td>13</td>
</tr>
<tr>
<td>2.1.2 Multiple Issue</td>
<td>15</td>
</tr>
<tr>
<td>2.1.2.1 Superscalar Processors</td>
<td>15</td>
</tr>
<tr>
<td>2.1.2.2 Software Approaches</td>
<td>16</td>
</tr>
<tr>
<td>2.2 Memory hierarchy</td>
<td>17</td>
</tr>
<tr>
<td>2.2.1 Cache performance</td>
<td>18</td>
</tr>
<tr>
<td>2.2.1.1 Reducing cache miss penalty</td>
<td>20</td>
</tr>
<tr>
<td>2.2.1.2 Reducing cache miss rate</td>
<td>20</td>
</tr>
<tr>
<td>2.2.1.3 Reduce memory access time with prefetching</td>
<td>21</td>
</tr>
<tr>
<td>2.2.2 Memory Performance</td>
<td>21</td>
</tr>
</tbody>
</table>
III Optimizing the multigrid algorithm on two selected architectures 69
  9 2D Red-Black Gauss-Seidel smoother on the x86/x86-64 Architecture 71
    9.1 The x86/x86-64 architecture: An overview 71
    9.2 Test machine characteristics 72
      9.2.1 Arithmetic Performance 72
      9.2.2 Memory characteristics 72
      9.2.3 Cache characteristics 73
    9.3 Optimization techniques 75
    9.4 Results 78
    9.5 Validation of the Results 82
  10 3D Geometric Multigrid on the IA64 Architecture 87
    10.1 The IA64 Architecture: A Short Overview 87
      10.1.1 Instruction Formats 88
      10.1.2 Register Set 89
      10.1.3 Selected Special Concepts 89
    10.2 Optimization Techniques 95
      10.2.1 Loop Optimizations 95
      10.2.2 Optimizations on Algorithmic Level 97
    10.3 Results 104
      10.3.1 Smoother 104
      10.3.2 Overview: Results for serial smoother 108
      10.3.3 Parallelization of Smoother 108
      10.3.4 Results for the Optimized Multigrid Method 108
      10.3.5 Analysis of convergence behavior 110
    10.4 Validation of IA64 results 111

IV Conclusions and Future Work 115
  11 Conclusions and Future Work 117
    11.1 Conclusions 117
    11.2 Chip Multi Processors 118
    11.3 Related Work 119
    11.4 Future Work 119

V Appendix 121
  A History of Computer Architecture 123
CONTENTS

B Test platforms .......................... 131
  B.1 Pentium 4 .................................. 131
  B.2 Athlon 64 .................................. 132
  B.3 Core 2 .................................. 133
  B.4 Itanium2 .................................. 133

C Results of the Dime Codes ................. 135
  C.1 Description of Different Dime Versions ........ 135
  C.2 Results .................................. 136

D x86 Instruction Set Architecture .......... 143
  D.1 Register set ................................ 143
  D.2 Instruction set ................................ 143
  D.3 Application Binary Interface ............... 151

E x86-64 Instruction Set Architecture ....... 153

F Introduction to gas assembler ............... 155

G Motivation of the test problem: Heat Equation 159

H German Parts .................................. 163

I Curriculum Vitae ............................... 165
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Instruction Pipeline</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Memory hierarchy of the Intel Nehalem processor</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>Memory technology</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>Numbering of the unknown vector</td>
<td>40</td>
</tr>
<tr>
<td>5.2</td>
<td>Coefficient matrix structure for Poisson equation</td>
<td>40</td>
</tr>
<tr>
<td>5.3</td>
<td>Coefficient matrix structure for Poisson equation in Red Black ordering</td>
<td>41</td>
</tr>
<tr>
<td>9.1</td>
<td>Measured cache bandwidth</td>
<td>74</td>
</tr>
<tr>
<td>9.2</td>
<td>Measured cache triad bandwidth</td>
<td>75</td>
</tr>
<tr>
<td>9.3</td>
<td>Separation of black and red values</td>
<td>76</td>
</tr>
<tr>
<td>9.4</td>
<td>Layout of red and black arrays x=9</td>
<td>77</td>
</tr>
<tr>
<td>9.5</td>
<td>Discrete five point stencil</td>
<td>77</td>
</tr>
<tr>
<td>9.6</td>
<td>Global update scheme for blocking technique</td>
<td>79</td>
</tr>
<tr>
<td>9.7</td>
<td>Fusion of two full iterations with a 8x2 block in 2D</td>
<td>79</td>
</tr>
<tr>
<td>9.8</td>
<td>Results for optimized Dime Fortran77 codes</td>
<td>81</td>
</tr>
<tr>
<td>9.9</td>
<td>Standard assembler implementation with optimized data layout</td>
<td>83</td>
</tr>
<tr>
<td>9.10</td>
<td>Streaming assembler implementation</td>
<td>84</td>
</tr>
<tr>
<td>9.11</td>
<td>Blocked assembler implementation</td>
<td>85</td>
</tr>
<tr>
<td>10.1</td>
<td>Branch instruction algorithm for br.ctop</td>
<td>94</td>
</tr>
<tr>
<td>10.2</td>
<td>Prefetching with different access patterns</td>
<td>97</td>
</tr>
<tr>
<td>10.3</td>
<td>Standard version of temporal blocking of two iterations</td>
<td>101</td>
</tr>
<tr>
<td>10.4</td>
<td>Temporal blocking of two iterations with additional blocking planes</td>
<td>101</td>
</tr>
<tr>
<td>10.5</td>
<td>Addressing problems</td>
<td>105</td>
</tr>
<tr>
<td>10.6</td>
<td>Memory layout optimizations</td>
<td>105</td>
</tr>
<tr>
<td>10.7</td>
<td>Loop optimizations</td>
<td>106</td>
</tr>
<tr>
<td>10.8</td>
<td>Blocked implementation</td>
<td>107</td>
</tr>
<tr>
<td>10.9</td>
<td>Extended Prefetching and blocked smoother</td>
<td>107</td>
</tr>
<tr>
<td>10.10</td>
<td>Specific padding</td>
<td>108</td>
</tr>
<tr>
<td>10.11</td>
<td>Overview optimized serial smoother</td>
<td>109</td>
</tr>
<tr>
<td>10.12</td>
<td>Parallelization of smoother</td>
<td>109</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

10.13 Overview of results for parallel smoother ........................................... 110
10.14 Runtimes for a single V-Cycle with a $513 \times 513 \times 513$ grid .................. 111
10.15 Convergence behavior on a $257 \times 257 \times 257$ grid .......................... 112
10.16 Comparison of optimized Smoothers in 3D on different architectures .......... 113

A.1 Electronic Numerical Integrator and Calculator (ENIAC) .................. 125
A.2 Electronic Delay Storage Automatic Calculator (EDSAC) .............. 125
A.3 Von Neumann Architecture ............................................................... 125
A.4 IBM 360 ......................................................................................... 126
A.5 CDC 6600 ....................................................................................... 126

C.1 Plain unblocked 2D dime versions ...................................................... 136
C.2 Version rb4 ...................................................................................... 137
C.3 Version rb5 ...................................................................................... 138
C.4 Version rb6 ...................................................................................... 139
C.5 Version rb7 ...................................................................................... 140
C.6 Version rb8 ...................................................................................... 141
C.7 Version rb9 ...................................................................................... 142

D.1 Stack Layout ...................................................................................... 152
# List of Tables

2.1 Technical data of common memory technologies ........................................... 23
6.1 Arithmetic peak performance ........................................................................ 49
6.2 In Cache Peak Performance: L1 ................................................................. 50
6.3 In Cache Peak Performance: L2 ................................................................. 50
6.4 Memcpy: Baseline ...................................................................................... 51
6.5 Memcpy: Non temporal stores ..................................................................... 52
6.6 Memread: Software Prefetching ................................................................. 52
6.7 Memcpy: Software Prefetching ................................................................. 53
7.1 Overview memcpy results ............................................................................ 60
7.2 Overview arithmetic performance .............................................................. 61
8.1 Compiler code analysis ............................................................................... 64
8.2 Compiler Flags .......................................................................................... 65
8.3 Overview Prefetching results ...................................................................... 67
8.4 Software prefetch effect with different vector lengths ............................... 68
9.1 Measured memory bandwidth ..................................................................... 73
10.1 Instruction Categories on the IA64 Architecture ....................................... 88
10.2 Estimate of flops and memory accesses per unknown on one grid level ..... 99
10.3 Distribution on the components for a V(2,2)-cycle .................................... 99
10.4 Optimized Multigrid Algorithm for 3D in Pseudo Code ............................ 102
10.5 Estimate of flops and memory accesses per unknown on one grid level with optimization ................................................................. 103
10.6 Distribution on the components for a optimized V(2,2)-cycle .................... 103
A.1 Overview: Milestone machines .................................................................. 129
D.1 Registers on x86 ...................................................................................... 146
D.2 Instruction types ....................................................................................... 150
D.4 Instruction list x86 .................................................................................. 146
D.5 Instruction list SSE .................................................................................. 149
D.1 Registers on x86 ...................................................................................... 150
LIST OF TABLES

D.3 Addressing modes .................................................. 150
Listings

6.1 Basic Triad benchmark in C ............................................. 48
6.2 Peak performance benchmark for the Pentium 4 ....................... 49
6.3 Algorithm Block Prefetching for vector triad ....................... 52
7.1 memcpy implemented in assembler using Complex Instruction Set Computing (CISC) instructions ...................................... 55
7.2 memcpy implemented in assembler using Reduced Instruction Set Computing (RISC) instructions .................................. 55
7.3 memcpy implemented in assembler using Streaming SIMD Extensions (SSE) instructions ................................................. 56
7.4 memcpy implemented in assembler using SSE instructions and non temporal stores ......................................................... 57
7.5 memcpy implemented in assembler using SSE2 instructions and non temporal stores ....................................................... 57
7.6 memcpy implemented in assembler using SSE2 instructions and non temporal stores with prefetch instructions .................. 58
7.7 memcpy implemented in assembler using SSE2 instructions and non temporal stores with prefetch instructions and usage of preloading ......................................................... 58
7.8 RISC data load .................................................................. 60
7.9 CISC data load .................................................................. 60
8.1 Data local iterative methods (DiME) blocked loop construct (Fortran 77) ................................................................. 63
8.2 Vector triad in C ................................................................. 65
8.3 Applying pragmas in C .......................................................... 65
8.4 Vector Triad: Compiler generated assembler code .................... 66
8.5 Applying compiler intrinsics .................................................... 66
8.6 Algorithm Block Prefetching .................................................. 67
10.1 If else branch .................................................................... 89
10.2 If else branch in x86 assembler (Conditional Branching) ......... 89
10.3 If else branch in simplified IA64 assembler (Conditional Branching) ................................................................. 90
10.4 For branch in C ................................................................. 90
10.5 For loop in x86 assembler ...................................................... 90
10.6 Unrolled for loop in x86 assembler ....................................... 91
Acronyms

**ABI** Application Binary Interface

**ALU** Arithmetic Logic Unit

**API** Application Programming Interface

**CISC** Complex Instruction Set Computing

**CMP** Chip Multi Processor

**CPI** Cycles per Instruction

**CPU** Central Processing Unit

**DiME** Data local iterative methods

**DRAM** Dynamic Access Memory

**EDVAC** Electronic Discrete Variable Computer

**EDSAC** Electronic Delay Storage Automatic Calculator

**ENIAC** Electronic Numerical Integrator and Calculator

**EPIC** Explicitly Parallel Instruction Computing

**FPU** Floating Point Unit

**HPC** High Performance Computing

**ILP** Instruction Level Parallelism

**ISA** Instruction Set Architecture

**LES** Linear Equation System

**MFlops** Mega Floating Point Operations per Second

**MMX** Multimedia Extension

**NUMA** Non Uniform Memory Architecture
ACROMYMS

**PDE** Partial Differential Equation

**RAW** Read After Write

**RISC** Reduced Instruction Set Computing

**SIMD** Single Instruction Multiple Data

**SSE** Streaming SIMD Extensions

**TLB** Translation Lookaside Buffer

**VLIW** Very Long Instruction Word

**WAR** Write after Read

**WAW** Write after Write
Chapter 1

Introduction

1.1 Scope

This work contributes to the field of optimization for iterative numerical algorithms operating on large, spatial datasets. It must be seen as a continuation of the PHDs of [Wei01] and [Kow04] with special focus on recent developments in computer architecture. The following areas are addressed in this work:

- Analysis of recent developments in mainstream processors with regard to optimization
- Analysis of the relationship between high level language, instruction set architecture and micro architecture of the underlying hardware
- Architecture specific optimization techniques for iterative linear equation solvers on structured data sets

The term optimization is always used in this thesis in the context of runtime optimizations.

1.2 Motivation

There is a broad range of scientific applications, where the available computational power limits the significance of the results. Among those are computational fluid dynamics problems or the simulation of large particle systems in molecular dynamics. As the computational power increases also the problems get more demanding. Today global scale multi physics simulations get in the scope of research. The increase in computational power is mainly based on the use of massively parallel machines. While there is a factor of around 17390 between the fastest machine on the first Top 500 Supercomputing list [MSDS08] released 1993 and today’s fastest machine in this list (June 2008), the single processor Linpack performance between e.g. the CDC Cyber 205 (1983) [Wik08a] and today’s fastest Vector processor NEC SX8R [Wik08b] differs only by a factor of around 43. This indicates that single Central Processing Unit (CPU) performance is not important, because the performance is resulting from the use of more processors. To reach high efficiencies on a large parallel machine it is indispensable that the code
also has a high single CPU performance [GH01]. Large scale parallel applications show this, as the results in [GH01] using highly optimized code reaching a nearly optimal single CPU efficiency. Research [HLW00] shows that in the limits of Amdahls Law massively parallel algorithms are single node performance limited. In [SSB+08] it is shown on the example of the massively parallel IBM BlueGene machine that the use of highly optimized libraries, providing excellent single node performance, is one key to success for good overall parallel results.

For many scientific codes the initial sustainable single node efficiency on modern processors is disappointing. The reasons for that are manifold. The processors used in todays massively parallel machines are often commodity processors not designed for one specific class but for a wide range of applications. While for our class of algorithms a vector machine is well suited, this processor type is rarely available. The processor design which proved to satisfy many requirements best are superscalar, pipelined processors with a deep memory hierarchy. Most of todays mainstream processors share this basic design. This type of processor is not optimal for our class of algorithms, where few computations are executed on large data sets with a uniform data access pattern. In this setting the Achilles heel of stored program computers (also referred to as von Neumann Architecture), the single bus data access, limits the performance.

This work is a continuation of the work done as part of the DiME project funded by the German Research Foundation, research grants Ru 422/7-1,2,3,5. The previous two PHDs which evolved as part of the DiME project named the high memory bandwidth requirements as the major bottleneck for our class of algorithms and developed optimization techniques to overcome these limitations and reach reasonable efficiency on common processors. The first thesis finished in 2001 [Wei01] developed and analyzed source code transformations to increase in cache data reuse. The second thesis finished 2004 [Kow04] besides other issues refined these techniques and applied them to selected algorithms to prove their effectiveness. As a result of this work source to source code transformations of the data layout access were introduced. These techniques were applied to grid based iterative numerical algorithms. As an outcome the effectiveness of these techniques could be shown with a significant performance increase. On classical RISC CPUs (DEC Alpha architecture) the speedup for a 3D iterative smoother algorithm was more than a factor of 2. On the Intel Pentium 4 processor these optimized codes performed disappointing with a speedup of around 30 % reaching only around 6 % of theoretical arithmetic peak performance [Kow04]. These results indicated that for an unknown other reason the optimization techniques did not work as intended. The analysis of the hardware performance counter data showed that while the cache misses were reduced significantly the branch prediction misses increased. The suspicion was that the complicated loop structures increase the number of branch prediction misses and also prevent the hardware prefetch unit present in the Pentium 4 to work effectively and by that decreases overall performance. Another suspicion was, that the small TLB of the Pentium 4 could cause many TLB misses, especially in the 3D case, where a large amount of memory is accessed involving many memory
pages at the same time. This suspicion was also supported by hardware performance counter measurements and the success of a TLB optimized matrix multiplication by K. Goto [GG02] on the Pentium 4.

The state of the art at the beginning of this thesis was, that memory bandwidth limitations can be overcome for selected classes of algorithms by applying cache blocking techniques and adopted data structures, but under the constraint that on the recent generation of processors other influences prevent satisfying results. At this point it was clear that the compiler has a large influence on the resulting performance. Still it was never questioned or analyzed systematically on instruction code level, if the generated instruction code quality is the reason for the low efficiency. After recompiling the optimized smoother codes in 2D and 3D with recent versions of the Intel compiler and evaluating the benchmarks again on the Pentium 4 processor the same program code showed a much improved performance compared to previous results with older compiler versions. The source code optimizations worked and the results were equal compared to the results on the DEC Alpha processors. This proved that the influence of the compiler is large, even that large that the benefit of source code optimizations can be undone. A comparison of the generated instruction code revealed that previous compilers were not able to generate efficient code for the nested loop constructs found in the optimized implementations. The compilers also failed to do an effective addressing of the multi dimensional fields and an effective register scheduling. On the x86 architecture with its limited number of eight general purpose registers, efficient addressing and register scheduling is much more difficult in comparison to RISC architectures with typically many more general purpose registers. These results encouraged us to do more research in order to overcome the problems on modern architectures and reach higher efficiency also there.

This thesis analyzes the influence on performance of scientific codes from the high level language code over the compiler influence to the instruction code and the ISA influence in general down to the underlying micro architecture. Based on the insights provided by this analysis refinements and extensions of the optimization techniques developed in the previous works will be introduced to achieve a higher efficiency on present micro architectures.

In the following sections a bottom up overview what changed on recent modern micro architectures is given in comparison to previous ones. Starting from the processors over the instruction set architecture to code optimization techniques possible influences on performance are introduced.

1.3 Developments in Computer Architecture

The Alpha architecture was one of the first mainstream implementations of an out of order execution superscalar processor with a deep cache hierarchy. Its RISC instruction set was well suited for efficient code generation by the compiler. In the early nineties it had a significant technology advantage against the mainstream x86 processors. At this time the micro archi-
tectures concentrated on superscalar pipelining to reach higher arithmetic performance. In a next step out of order speculative execution was introduced together with sophisticated branch prediction algorithms. These techniques together with rapid increases in clock speeds of the cores resulted in huge improvements of arithmetic performance of general purpose processors. In the late nineties so called multimedia instruction set extensions were introduced to support a higher performance for multimedia applications. These applications, as e.g. image processing and video encoding, are often using streaming algorithms, where large amounts of data are processed independent from each other. An example for such a multimedia extension is the Multimedia Extension (MMX) introduced with the Intel Pentium processor and the SSE and SSE2 instruction set extensions introduced with Pentium 3 and Pentium 4 processors. These extensions introduced the SIMD execution model. All previous optimizations as the use of a memory hierarchy were designed to work transparently. These new optimizations required an adopted code. While the SIMD paradigm is well suited for many applications, the compilers in the beginning had no support for these instructions and forced the software developers to program in assembly to benefit from the extensions [BJER98]. To use these new instructions the compiler has to auto vectorize the loops, which puts high demands with regard to dependency analysis and data alignment on the compiler [KA02]. While at the beginning these instructions were not implemented efficiently starting with the Pentium 4 the micro architectures were optimized to execute these instructions. To get a reasonable performance it was now necessary to use these new instructions. Starting with the x86-64 ISA the 64 bit enabled successor of the x86 ISA the SIMD instruction set extensions replace the x86 Floating Point Unit (FPU) instructions.

The x86-64 ISA is an evolution of the x86 ISA, removing weaknesses as the limited number of general purpose registers and the register stack for floating point calculations, while retaining backward compatibility. In contrast to this Intel introduced together with Hewlett-Packard the revolutionary new IA64 ISA. IA64 addresses many known issues of established ISA. It introduces Explicitly Parallel Instruction Computing (EPIC), which is based on Very Long Instruction Word (VLIW) concepts, to increase exploitable Instruction Level Parallelism (ILP). In IA64 the optimization task is shifted to the software side, reducing complexity on the hardware and allowing to put more basic hardware resources on the chip, as e.g. registers and caches. This puts a high pressure on the compiler: If code is not optimized there is no hardware logic to attenuate this and performance can be low. The only available implementation of the IA64 ISA is the Intel Itanium 2 processor using an in order execution architecture.

An important technique introduced by all modern processors today is data prefetching. The x86/x86-64 architectures have a dedicated hardware prefetch unit. This is a unit on the processor trying to detect a data access pattern in order to trigger loads in advance and thereby hide data access latencies. Data prefetching favors long running inner loops which contradicts to the small inner loop lengths introduced with a blocking optimization. This fact demands for a prefetching aware blocking strategy.
INTRODUCTION

Overall it can be said that the furious increase in arithmetical single core performance due to clock increase has come to an end and other strategies are necessary to further increase performance. This can also be seen in the fact that recent micro architecture improvements target the memory subsystem. Recent developments are the introduction of Chip Multi Processor (CMP) architectures with specialized arithmetic processing units (e.g. Cell architecture or graphics processor architectures) or multiple cores on one chip sharing parts of the memory hierarchy. These new architectures offer many new opportunities but come with the price of a higher overall complexity of the hardware and the software.

These new developments are not treated explicitly in this work. Still hardware aware optimization techniques are more and more important to utilize modern processors as e.g. the Cell processor and recent multi core desktop processors efficiently.

1.4 The Hardware Software Interface

The instruction set and its hardware implementation decide how difficult it is for a compiler to translate high level language code as C or Fortran into an efficient sequence of machine instructions. During execution the interaction between the instruction code and the executing hardware implementing the instruction set solely decide about performance. All techniques introduced with the multimedia extensions on x86 architectures make this task more difficult. In many cases it is not possible for the compiler to extract enough information from the high level language to apply optimizations. The classical RISC architectures were designed to suit the needs of compilers. High level language and ISA were designed in an integrated fashion. At the same time RISC ISAs suited the new breed of pipelined superscalar processors creating an integrated overall concept. Since that time new additions to the instruction sets broke this integration, making it more difficult for compilers to create efficient code. Also on new architectures like IA64, which were designed from the beginning to shift optimization tasks to the compiler, it is not possible for the compiler to produce reliably efficient code. The experience in this thesis shows that there can be a significant performance loss by relying on compiler generated code. One main aspect of this work is to learn the details of the covered instruction set architectures together with their hardware implementations to get the best possible result with regard to performance.

1.5 The Optimization Process

To achieve reliably good results with hardware specific optimizations it is necessary to introduce a systematic approach for the optimization process. A step often underestimated in this context besides the code analysis is the machine analysis. In this thesis micro benchmarks are used as a tool to learn what a particular machine demands and is capable of. This involves a thorough knowledge of the computer and instruction set architecture. Only based on this knowledge a reliable judgment with regard to code efficiency and optimization opportunities
can be formed.

The final aim of optimization efforts in our area must always be runtime. Especially in the High Performance Computing (HPC) community often Mega Floating Point Operations per Second (MFlops) and hardware performance counter measurements are in the focus of research. This can be misleading and lower the significance of the results. These values are important to compare results and judge how efficient an algorithm runs on a particular machine. Hardware performance counter measurements are well suited to get additional information in the analysis phase but should never be an end in itself. The main motivation and target for the optimization must be runtime.

1.6 Outline

Part I of this thesis covers the background knowledge relevant for this work. In Chapter 2 a basic introduction to computer architecture is given. A preparation to this chapter is the short overview of the history of computer architecture in Appendix A. The hardware software interface, the ISA, is covered in Chapter 3. Annotations and examples with regard to code optimizations is given in Chapter 4. The numerical theory for the solution of our model problem is covered in Chapter 5. A physical motivation for our model problem with a physical and mathematical derivation can be found in Appendix C. In Part II an analysis and examples of influences of the hardware architecture (Chapter 6), ISA (Chapter 7) and the compiler (Chapter 8) on performance are given. Part III presents results for the optimization of the Red-Black Gauss-Seidel smoother method on the x86/x86-64 architecture (Chapter 9) and an optimized geometric multigrid method on the IA64 architecture (Chapter 10). Conclusion and suggestions for future work can be found in Part IV.
Part I

Background
Chapter 2

Computer Architecture:
Understanding the machine

In this chapter an overview of the for our application relevant computer architecture is given. Not every aspect can be treated in depth, still it is attempted to give a general overview. One can distinguish hardware optimization techniques into those, who are supposed to work without affecting the programming and those which rely on an adopted software to work effectively. Also those techniques not relying on adopted software may benefit from a hardware aware implementation. The reason for this is, that these techniques as e.g. caches or hardware prefetching make assumptions to the software. Not every software fulfills these assumption causing a processor not to work as effective as intended. Hardware aware programming can dramatically increase the efficiency of the hardware. A thorough understanding of computer architecture is crucial to be able to write efficient code. Because it is helpful to know the historical background in order to understand the developments in computer architecture in Appendix A a historical overview about main developments in computer architecture is given. For reference and further reading we refer to common textbooks on computer architecture (e.g. [HP03]). To better understand the x86 assembler code listing in this chapter it is useful to read the introduction to the x86 ISA in Appendix D. In Section 2.1 techniques to improve arithmetic performance are described. A description of the memory hierarchy can be found in Section 2.2.

2.1 Instruction Level Parallelism

A key technique to improve the arithmetic performance in processors is to overlap the execution of instructions. The overlap of instructions is also called instruction level parallelism (ILP), because the instructions are evaluated in parallel. The major implementation for improving ILP is pipelining (2.1.1). Outside the memory subsystem, the effective operation of the pipeline is usually the most important factor to determine the performance of a processor. There are basically two methods for increasing the potential
for ILP. First you can increase the depth of the pipeline to overlap more instructions. In this setting the clock cycle can be shorter because less is done in each stage. Another approach is to replicate internal components of a computer so it can launch multiple instructions in every pipeline stage. This technique is called multiple-issue. There are two ways to implement multiple-issue: static multiple issue (explained in section 2.1.2.2) and dynamic multiple issue (described in section 2.1.2). In the following section some elementary knowledge about instruction set architecture is required. If these topics are completely new to the reader it is recommended to read Section 3.2 in advance.

2.1.1 Instruction Pipelining

A pipeline is working similar to an automobile assembly line. There are several steps completing different parts of an instruction. Each step is called pipe stage. A main property of a pipeline is its throughput or number of instructions per time to exit the pipeline. Because the stages are hooked together they must operate at the same speed. On a processor everything is synchronized through the processor clock. The time required between moving an instruction down one step is a processor cycle. The length of a processor cycle is then determined by the time required for the slowest pipe stage. A key goal for designing a pipeline is therefore to balance all stages. If the stages are perfectly balanced, then the time per instruction is:

\[
\text{Time per instruction on unpipelined machine} = \frac{\text{Number of pipe stages}}{(2.1)}
\]

The basic mechanism of a pipeline is illustrated in figure 2.1. The benefit from pipelining can be seen as a combination of two factors. Reducing the number of clock cycles needed for one instruction and reducing the time needed for one clock cycle. But pipelining does not reduce the overall clock cycles needed for each individual instruction to complete. The key is that by overlapping the execution of instructions each cycle one instruction finishes and the average time for each instruction is reduced. Overall this results in reducing the average execution time per instruction. The measure for how efficient a pipeline works is clock cycles per instruction (CPI). If a processor needs several clock cycles for an instruction pipelining will make it faster by reducing the CPI. If processor already needs only one cycle per instruction then adding (more) pipelining and therefore reducing the time needed for one pipeline stage will reduce the time of one clock cycle, because it is possible to clock the processor higher.
Pipelining tries to exploit parallelism in a sequential instruction stream. It has the advantage that unlike other speedup techniques the programmer is not forced to adapt the program code to benefit from it. The first step for designing a pipeline is to split up an instruction into stages. As an example this is shown for the classical 32 bit RISC integer pipeline. Each instruction is split up into the following subtasks, where a task can be completed in one clock cycle.

1. **Instruction fetch cycle (IF)**
   Fetch current instruction from memory. Update program counter (PC) to the next sequential PC by adding 4 to the PC.

2. **Instruction decode/register fetch cycle (ID)**
   Decode the register and read the registers corresponding to register source specifiers from the register file.

3. **Execution/effective address cycle (EX)**
   Perform one of three function depending on the instruction type.
   - Memory reference: The ALU adds the base register and the offset to form the effective address
   - Register=Register ALU instruction: The ALU performs the operation specified by the ALU opcode on the values read from the register file
   - Register immediate ALU instruction: The ALU performs the operation specified by the ALU opcode on the first value from the register file and the immediate

4. **Memory access (MEM)**
   If instruction is a load, perform a read using the effective address from the previous cycle. If it is a store write data from the register using the effective address.

5. **Write-back cycle (WB)**
   Write the result into the register file, whether it comes from the memory subsystem (for a load) or from the ALU (for an ALU instruction)
CHAPTER 2

This is already an example for a typical RISC pipeline. Above descriptions allow to derive how an ISA should look like to be suited for pipelined execution. All instructions must have the same length, this makes it much easier to fetch it in the first stage and decode it in the second stage. Second it is favorable to only have few instruction formats with source register files located at the same place in each instruction. This eases reading from the register file in the second stage. Memory operands only appear in loads or stores. With that restriction the EX stage can be used to calculate the memory address and then access memory in the following stage. If there are instruction allowed which operate on the operands in memory, an additional address stage would be necessary. And operands must be aligned in memory so that data can always be transferred between memory and processor in one request.

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called pipeline hazards. While there exist a number of hardware techniques to face pipeline hazards, many hazards can be avoided in advance when implementing the software. The presence of data and control dependencies, which can become hazards, are the primary limitations on how much parallelism can be exploited in an instruction stream.

1. **Structural Hazards:**
   - The hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.

2. **Data Hazards:**
   - A instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.

3. **Control Hazard:**
   - The pipelining of branches and other instructions changing the Programming Counter (PC).

A pipeline hazard can make it necessary to stall the pipeline until the hazard cleared. In that time no new instructions can be fetched or issued meaning that stages keep unused for that period. This is is called to insert bubbles in the pipeline.

2.1.1.1 **How to prevent data hazards**

A typical data hazard occurs, if one instruction depends on a result of a previous one, that is still in the pipeline. Consider the example:

A  add eax, ecx  
B  sub eax, ecx

Without intervention this would cause a severe pipeline stall, because it needs several pipeline stages until the result of A is written back to the register file. The point is that the result is already available inside the pipeline. The solution to this problem is to provide extra hardware, that inputs the result of instruction A directly to instruction B. This process is
called forwarding or bypassing. Many data hazards can be avoided by forwarding. Many data dependencies can be removed by renaming the variables. Such data dependencies are also called name dependencies. The two types of name dependencies are:

1. An anti dependence: B writes to an register or memory that A reads.

2. An output dependence: Occurs when A and B write the same register or memory location.

True data hazards can be classified as one of three types, depending on the order of read and write accesses in the instructions. Let’s consider two instructions A and B, where A occurs before B in the instruction stream. The possible data hazards are:

- RAW (read after write)
  B tries to read a source before A writes to it.

- WAW (write after write)
  B tries to write an operand before it is written by A. The writes end up in the wrong order.

- WAR (write after read)
  B tries to write a destination before it is read by A, so A incorrectly gets the new value.

If there is an unavoidable hazard, then the hazard detection hardware stalls the pipeline. No new instructions are fetched or issued until the dependence is cleared. To overcome this the compiler can try to schedule instructions in order to avoid the hazard. This is called compiler or static scheduling.

Such a static approach is also called in-order execution. An in-order processor executes the instructions in the same order they are present in the instruction stream. To avoid a pipeline hazards besides forwarding the instructions have to be reordered already at compile time.

With dynamic scheduling the hardware rearranges the instruction execution to reduce the stalls while maintaining data flow. The drawback is a significant increase in hardware complexity. Issue in order but execution out-of-order. Out-of-order execution offers some advantages but also introduces many new problems. Dynamic scheduling can handle cases when dependencies are not known at compile time. One advantage is, that it simplifies the compiler.

Moreover it allows to run code which was compiled with one pipeline in mind to run more or less efficiently on a different pipeline. This comes at the price of a much higher hardware complexity. The high complexity makes it very difficult to understand how the code is scheduled and executed and reduces efforts to improve scheduling to trial and error.

2.1.1.2 How to prevent control hazards

Control hazards occur if it is necessary to make a decision based on the result of one instruction while others are executing. Control hazards are also often called branch hazards. If there is a branch instruction it is necessary to fetch the instruction following the branch on the very
next clock cycle. But the pipeline cannot know what the next instruction should be, since it only just received the branch instruction from memory. One possible solution is to stall the pipeline until the pipeline determines the outcome of the branch and knows what instruction address to fetch from.

An better solution is to predict the outcome of the branch and therefore prevent bubbles. Branch prediction assumes an outcome of a branch and proceeds from that assumption rather than waiting for the actual outcome. When the guess was wrong, the pipeline control has to ensure that the instructions following the wrongly guessed branch have no effect and must restart the pipeline from the proper branch address.

The simplest scheme of that kind is to treat every branch as not taken. The pipeline just continues as if the branch was not taken. It has to be taken care not to change the processor state until the branch outcome is definitely known. This scheme is called predicted-not-taken, of course also a scheme predicted-taken is possible. For each of these schemes the compiler can improve performance by organizing the code so that the most frequent path matches the hardware choice. Another static scheme is called delayed branch. It introduces a successor instruction after each conditional jump instruction. This successor is the branch delay slot.

There are as many branch delay slots as many delay cycles a branch takes on a given pipeline.

For a classical five stage RISC pipeline there is one delay slot. Now the compiler fills the instruction after the branch with a instruction independent of the branch outcome. If the branch turns out to be not taken the instruction after the branch delay instruction is executed, if the branch is taken execution continues at the branch target.

All schemes up to now were static, meaning that the action taken does not depend on the dynamic behavior of the branch. Better schemes use hardware to dynamically predict the outcome of a branch. The prediction depends on the behavior of the branch at run time and will change if the branch changes its behavior during execution. The need for a more sophisticated branch prediction arises from todays deeply pipelined, multiple issue processor. With these processors first of all a branch arrives much faster than before and if a stall occurs the penalty is by far bigger. As a consequence on such processors control dependencies often become the limiting factor for performance. The problem when covering dynamic exploitation of ILP is, that all these techniques are coupled and connected to each other. A thorough coverage is therefore difficult in a compressed introducing overview. The following remarks can only act as an introduction to the basic terms. For a deeper coverage of these more advanced topics we refer to the standard text books on computer architecture as [HP03].

The simplest dynamic schemes use a branch-prediction buffer. This small memory is indexed by the lower portion of the address of the branch instruction and contains a bit that says whether a branch was recently taken or not. While this is a local scheme which only can store the outcome of one particular branch more sophisticated branch-prediction buffers also take the behavior of other branches into account. This technique is called correlating predictors. By combining global and local predictors and adaptively combine them by a selector you get a so called tournament predictor used e.g. in the Alpha 21264 processor. While these techniques
predict the outcome of a branch, they do not reduce the branch penalty. To maintain a high bandwidth instruction stream it is necessary to know from what address to fetch as early as possible. This can be accomplished by using a branch-target buffer. With this technique the next instruction address is already known, even before the branch instruction is decoded. In the effort to execute many instructions in one cycle today’s processors use an integrated instruction fetch unit with an integrated branch prediction and instruction prefetch.

2.1.2 Multiple Issue

In the previous chapter pipelining was used to evaluate instructions in parallel to increase execution speed. One way to increase instruction throughput even more is a technique called multiple issue. With this technique it is possible to launch multiple instructions in one clock cycle and reach CPI values less than one. As could be seen in the previous section it is already difficult to extract enough parallelism while avoiding hazards to keep a normal pipeline busy. Multiple issue puts even more pressure on the execution logic to keep the pipeline busy as more instruction can be executed in one cycle. For a processor this means it has to replicate its execution units. Typically modern processors can issue three to eight instructions every clock cycle. Again it is necessary to decide how to distribute the work of exploiting all existing ILP among hardware and software. As with pipeline scheduling a static approach at compile time has the advantage of a global view on the code allowing to apply a better optimization for a larger instruction window. On the other hand many decisions can only be done at runtime and the responsibility is large for the compiler to produce efficient code. The most prominent example for a static multi issue processor is the Itanium 2 processor implementing the IA64 ISA. Because the pressure to provide enough instructions to be issued on a multi issue processor is so large the analysis of the instruction window often provides not enough information to decide on the base of real knowledge. One of the most important methods of exploiting more ILP is therefore speculation. Speculation allows the compiler or processor to guess about properties of an instruction, to enable execution to begin for other instructions that may depend on the speculated instruction. Speculation may be wrong and a method has to be provided to check if the speculation was right or not and if it was wrong to back out or unroll the effects of the instructions that were executed speculatively. Speculation adds a lot of hardware complexity to a processor. It can improve performance significantly but decreases performance when done carelessly.

2.1.2.1 Superscalar Processors

Strictly spoken the term superscalar processors refers to a processor having more than one ALU deciding whether zero, one or more instructions can issue in a given clock cycle. In a simple superscalar processor instructions issue in order, leaving the work of finding ILP to the compiler. Most superscalar processors implement beyond dynamic issue decision so called dynamic pipeline scheduling. Dynamic scheduling allows the processor to reorder the
instructions in order to avoid stalls. The best benefit and state of the art superscalar processors support dynamic pipeline scheduling often called out of order execution together with speculative execution and dynamic branch prediction. This strategy of putting the burden of finding ILP on the processor side can be seen critically. The hardware implementation adds a lot of complexity to the logic on the processor and the benefit is not always present. For the programmer on those processors it is difficult to understand what is actually happening on the processor. Recent generations of processors tend to reduce the complexity in favor of other strategies, as larger caches or multiple simple processors on one die. The actual details and implementation issues for a superscalar processor are far beyond the scope of this thesis and not necessary for implementing efficient code. For further reading we refer to the classical text books on computer architecture (e.g. [HP03] or [HP05]).

2.1.2.2 Software Approaches

In contrast to the dynamic exploitation of ILP static approaches put the burden of finding ILP among instructions on the software side, hence the compiler or programmer. The amount of ILP, that can be exploited dynamically is limited. Even an issue of 3 instruction per cycle put a high pressure on the processor to fill these available slots. To exploit even more ILP a software approach has the advantage that it has a global view on the code. Also at compile time there is time and resources to apply more sophisticated optimizations not possible in a processor implementation. The disadvantage is that at compile time there is no runtime information available, especially for branches. An advantage of a software approach is that the core logic can be much simpler, allowing to use the available transistors in other ways. Even a statically issued superscalar processor has to check for dependencies between instructions. To avoid this the compiler could already format the instructions into an issue packet with multiple operations per instruction. The software guaranties that there are no dependencies inside the issue packet and the packet can be executes in parallel. This approach is also called VLIW (very long instruction word). This approach appealing to exploit much more ILP, than an superscalar dynamically scheduled processor could handle. The instruction packet needs to reflect the available execution units on the processor. Therefore only a limited number of instruction mixes are allowed within such a packet. To support more packet formats in order to give the compiler more flexibility in code generation these processors have a large amount of execution units. The problem is that for standard code it can be difficult to find enough parallelism to fill all available slots. To find large pieces of straight line code with enough ILP massive optimizations with loop unrolling and complicated instruction mixes are necessary. If the compiler is not able to fill a slot in an instruction packet, it needs to write an NOP in it, potentially wasting instruction code size. Often the NOP is still executed. This already points to the main disadvantages: Code size can be very large due to optimization efforts to get enough straight line code and due to unused slots, which could not be filled with sensible instructions. The limited number of instruction packet formats together with often further restrictions and with the massive amount of necessary ILP means a very high
complexity to the compiler. While once the processor is designed and manufactured, further improvements in technology needs an processor upgrade, with a software approach a processor benefits at once of technology advances made on the compiler side.

There are several possible optimizations to avoid the disadvantages of a static issue processor while obtaining the best possible ILP:

- Predication
- Software pipelined loops
- Enhanced software prefetching
- Explicit speculation

A more detailed description of these techniques is given on the example of the IA64 architecture in Section 10.

2.2 Memory hierarchy

Programmers desire an unlimited amount of memory. This contradicts the wish to have a memory fast enough to match the performance of the processor. Today the memory performance in terms of latency and bandwidth is way to slow to support peak performance for most algorithms acting on large data sets. Technically it is possible to create faster memory, but this is costly and it is not possible to have large amounts of this type of memory. A solution to this dilemma is the observation that many programs do not access all their code or data with equal probability. This fact is called the principle of locality. This principle states that you access a relatively small portion of your address space at any instance of time. There are two types of locality:

- Temporal locality:
  If an item is referenced it is likely to be referenced again soon.

- Spatial locality:
  If an item is referenced, items whose address are close will likely be referenced soon.

To take advantage of this principle of locality the memory on modern computers is implemented as a memory hierarchy. It consists of multiple levels of memory with different speeds and sizes. As the distance from the processor increases, the size of the memories and the access times increase.
The goal is to have as much memory as is available with the cheapest technology, while providing access at the speed offered by the fastest technology. The use of a hierarchical memory design already favors temporal locality. For organizational reasons the smallest amount of information transferred between levels of memory is a block or line, consisting of several data items. This favors spatial locality, because if one item in a block is accessed the whole block is transferred and access to another item in the block will be fast. The first implementation of a memory hierarchy design, which was the IBM 360/85 released 1969, these additional memory layers between processor and main memory were called caches, which means safe place to hide or store things. To construct a memory subsystem is a trade off between average performance gains and cost effectiveness.

### 2.2.1 Cache performance

There arise some basic questions in the design of caches. These are:

1. Where can a block be placed in cache?
   - Every block in memory can be placed at one position in the cache. Such a cache is said to be direct mapped.
   - A block can be placed anywhere in the cache. This is called fully associative cache.
   - If a block can be placed at a restricted set of places in the cache, the cache is called set associative.

2. Which block should be replaced on a cache miss?
   With direct mapped caches this choice is simple: There is no choice. Only one block can be checked for a hit and only this single block can be replaced. With fully associative or set associative placement there are many blocks that can be replaced. There are the following strategies on selecting which block to replace:
   - **Random**: Candidate blocks are selected randomly

![Figure 2.2: Memory hierarchy of the Intel Nehalem processor](image)
• **Least recently used** (LRU): To minimize the chance of throwing out a block which is needed soon, accesses to blocks are recorded. The block which has been unused for the longest time is replaced. This strategy again uses the principle of temporal locality for selecting a block to be replaced.

• **First in, First out** (FIFO): Because LRU is complicated to implement, FIFO tries to approximate LRU by choosing the oldest block rather than the LRU.

3. **What happens on a write?**

Most processors are optimized for reads. The usual strategy relies on the assumption that the dominant part of all memory accesses are reads. Still with Amdahls law in mind one cannot neglect the speed of writes. The problem with writes is that after a write into the cache, memory in lower levels of memory would have a different value from that in the cache. Cache and memory are said to be inconsistent. There are two strategies for writes:

• **Write through**: The information is written to the block in the cache and to the block in memory. Write through caches are easier to implement. The cache is always clean, so read misses never result in writes to lower memory. Another advantage is that the lower level has the most recent copy of data, simplifying data coherence. The major disadvantage of write through policy is, that the processor has to write stall during waiting for writes to complete. A common technique is to use a write buffer, which allows the processor to continue as soon as the data is written to the buffer.

• **Write back**: The information is written only to the block in the cache. The modified block is written to main memory only when it is replaced. To reduce the frequency of writing blocks back each block has a status bit, indicating whether a block was modified or not while in the cache. If it is clean (not modified) it is not written back on a miss. Write back has several advantages. Writes have the full speed of the cache memory and multiple writes within one block require only one write to lower level memory. Since not all writes go to memory it also saves memory bandwidth.

When a write miss occurs, there are two basic options. The block containing the miss is allocated and write misses act like read misses. Subsequent writes to the same address will only cause write misses for the first time. Or the block containing the miss is not allocated, meaning that write misses do not affect the cache. Subsequent writes to the same address with this *no-write allocate* strategy will always cause write misses. Write back caches often use write allocate hoping that subsequent writes are captured by the cache. Write through caches on the other hand often use a no-write allocate strategy, because if there are subsequent writes to the same address it still must be written to memory so nothing is to be gained.
Often the performance of a memory subsystem is measured in cache hit rate because it is independent of the speed of the hardware. A better measure for the efficiency of a memory subsystem is the average memory access time.

\[
\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}
\]

(2.2)

The average memory access time can be either measured in absolute time or in clock cycles, which enables comparisons between different platforms.

### 2.2.1.1 Reducing cache miss penalty

The most important technique to reduce cache miss penalty are multi level caches. By recursively applying the principle of locality new levels are added until the inner most level matches the speed of the processor while the outer level minimizes access to main memory. Write through caches are combined with a write buffer. There are two additional techniques connected to write buffers used to reduce cache miss penalty. The first is to give reads priority over writes. In the context of write buffers there could be a RAW hazard making it necessary for a read to wait until the write buffers are empty. A common alternative is to check the contents of the write buffers on a read miss and if there are no conflicts to continue with the read. Another common technique is to merge write buffers. This will use the memory more efficiently since multi word writes are usually much faster than writes performed one word at a time. Especially in the case of small direct mapped caches a technique called victim cache is used. A victim cache is a small fully associative cache, usually only a view entries large and is placed between two cache levels. If a block is discarded because of a miss it is placed in the victim cache. On a miss first the victim cache is checked for a hit. If there is a hit the victim block and the cache block are swapped. This can be seen as a kind of recycling because a block that was already fetched is reused.

### 2.2.1.2 Reducing cache miss rate

All misses can be sorted into three categories:

- **Compulsory**: Also called cold start misses. Occurring on the very first access to a block.
- **Capacity**: The cache cannot hold all blocks needed during execution of a program. Therefore blocks are discarded and later retrieved again.
- **Conflict**: If the block placement strategy is not fully associative conflict misses occur, because a block is discarded and later retrieved if too many blocks map to its set.

Optimizations include increasing the associativity to reduce conflict misses, increasing cache size to reduce capacity misses and make the blocks larger to reduce compulsory misses. In reality things are less simple and a designer has to have an eye on the whole system performance since a design change might have side effects or shift a miss of one class to another class.
Increasing the block size will reduce compulsory misses, but at the same time larger blocks increase miss penalty and since they reduce the number of blocks in a cache may increase conflict misses and capacity misses. As the overall strategy is to reduce miss rate and miss penalty the selection of the block size depends on the latency and bandwidth of the lower-level memory. With high latency and high bandwidth for example a larger block size might be favorable, because the cache gets more bytes per miss for only a small increase in miss penalty. Reducing capacity misses with larger caches has the drawback of longer hit time and higher cost.

2.2.1.3 Reduce memory access time with prefetching

With a regular, predictable data access pattern many cache misses can be prevented by issuing data reads in advance. Data prefetching can be either be done in hardware or in software. A hardware prefetch unit tries to detect regular data access patterns and trigger loads of the data in advance. The intelligence of such an approach is limited and usually only simple data stream patterns can be detected. Data prefetching can also decrease performance by prefetching unnecessary data which pollutes the cache and wastes bandwidth. These problems, also with regard to timing, can be prevented by issuing data prefetches inside the code. Desktop processors use both techniques to ensure a good performance for unoptimized codes and still allow software prefetching for more complicated cases not detected by the hardware prefetch unit. A pure software approach is e.g. implemented on the IA64 architecture. Data prefetching is of very high importance for the overall performance on modern architectures. For a programmer this means, that besides data access locality a data access pattern suited for hardware data prefetching, preferably long continuous data streams with constant stride, is at least of the same importance to achieve good performance on modern architectures.

2.2.2 Memory Performance

As already mentioned in the historic overview the memory is connected to the processor through a bus. This bus has a certain bus width and a clock with which it is driven as main specifications. The bus width governs how much data can be transmitted in one bus cycle. The clock signal can be seen as a peak signal with a rising and a falling edge. The clock is static and gives an upper bound how much data can be transfered over the bus. This data amount is calculated as effective bus bandwidth times the clock. This is also called peak memory bandwidth. Up to now the bus is described as a hose, water is filled in on one side and it comes out on the other. In reality the bus is a two way communication link between processor and memory controller. The processor has to gain access on the bus, then it sends a request for data. The RAM has to find the data and prepare it for transmission. Once the data is ready The RAM has to request control over the bus and transmit the requested data back to the processor. All these steps require some time. The total time is called memory latency. Memory latency is the time gap between a data request put on the bus and the requested
data coming back on the bus. In this time the valuable transport cycles are not filled with useful data and available memory bandwidth is not used. Read latency is effectively limiting the sustainable memory bandwidth. It is crucial to avoid these delays and keep the bus full with useful data. One of the first approached is to minimize the number of requests necessary to get a certain amount of data to the processor. As the caches are organized in cache lines which contain multiple words data is also send cache line wise over the memory bus. This is also called burst mode. In contrast to sending one word for each request the RAM sends a series of words over the bus for one request, one word in each bus cycle (for the case that the bus width is one word). Reducing memory latency will increase sustained memory bandwidth. To get best possible memory bandwidth as much data as possible has to be transmitted on a single data request. Older bus protocols only allow a single bus master access to the bus for a limited time. This is to ensure that no one blocks the bus. But this feature limits the maximum length of burst transmissions. Modern bus protocols support data streaming to allow longer burst mode transmissions. Again the techniques described here favor regular, sequential data access over a long vector. Applications with random single word data access characteristics will only achieve a small fraction of peak memory bandwidth because they pay the latency overhead for every single word requested. The first technique for better memory bandwidth is to use burst mode transmissions. What else can be done?

A next optimization is to either use a wider bus or to increase the clock of the bus to get a higher memory bandwidth. Increasing the bus clock not only increases the peak memory bandwidth but also decreases memory latency, because the same number of cycles take fewer time. Increasing the bus clock frequency unfortunately is usually more expensive than to widen the bus. A common technique to effectively double the bus bandwidth is to transmit data on the rising and on the falling edge of the clock signal. This technique is call double data rate (DDR) RAM, and can be seen as doubling the bus clock. Still the use of DDR RAM does not influence the read latency. That means a SDRAM with a two times higher bus frequency than a DDR RAM, which has the same peak memory bandwidth, has a higher sustainable memory bandwidth, because its read latency is smaller. A next optimization uses the fact that the memory is organized in banks. By interleaving memory access over the banks memory bandwidth is increased, since the effective bandwidth of every bank is added up. This technique is used on most modern platforms with dual channel RAM and excessively used on vector machines as e.g. the NEC SX8. A major benefit is only achieved together with a sequential memory access. The memory chip clock has not improved much over time. Improvements are coming through using dual channel RAM and by that doubling the bus width to 128 bit and even more fetching several words on each memory clock. From SDRAM to DDR SDRAM on one memory cycle not one but two words are transfered. While with DDR SDRAM the bus clock matched the memory clock with DDR2 SDRAM the bus clock is twice the actual memory clock. This enables that per memory cycle not 2 words but 4 words are fetched from several cells at the same time. So the number of words fetched per memory
cycle increased from one word in SDRAM to two words in DDR SDRAM to four words in DDR2 SDRAM. The next generation DDR3 SDRAM is just an evolution by fetching eight words per memory cycle. An overview to common memory technologies is given in Table 2.1 while Figure 2.3 illustrates how common memory types scale with bus clock. One improvement is not explicitly mentioned in Table 2.1. The on die memory controller introduced with the AMD Athlon64. This technique saves additional time otherwise needed by going over the mainboards northbridge chip to access memory resulting in lower read latencies (around 40-60 ns) and by that also enabling a higher sustained memory bandwidth.

![Figure 2.3: Memory technology](image)

<table>
<thead>
<tr>
<th>Memory type</th>
<th>bus clock</th>
<th>bus width</th>
<th>bandwidth</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPM DRAM 1987</td>
<td>asynchronous</td>
<td>32bit</td>
<td>-</td>
<td>35 ms</td>
</tr>
<tr>
<td>EDO RAM 1995</td>
<td>asynchronous</td>
<td>64bit</td>
<td>-</td>
<td>25 ms</td>
</tr>
<tr>
<td>SDRAM 1997</td>
<td>66 MHz</td>
<td>64bit</td>
<td>528 MB/s</td>
<td>260 ns</td>
</tr>
<tr>
<td>DDR SDRAM 2000</td>
<td>100 MHz</td>
<td>64bit (128bit dual channel)</td>
<td>1.6 GB/s (3.2 GB/s)</td>
<td>180 ns</td>
</tr>
<tr>
<td>DDR2 SDRAM 2004</td>
<td>200 MHz</td>
<td>64bit (128bit dual channel)</td>
<td>3.2 GB/s (6.4 GB/s)</td>
<td>124 ns</td>
</tr>
</tbody>
</table>

Table 2.1: Technical data of common memory technologies
2.2.3 Virtual memory

With virtual memory programs are linked using virtual addressing. This means they do not use physical addresses but use their private address space starting from any number, commonly zero. During runtime these virtual addresses need to be translated to physical addresses for hardware access to memory. There are several advantages to use such a scheme. On a computer there run several programs (or processes in an operating system sense) at any given time. As these processes only access a small amount of their address space at one time it makes sense to share a smaller amount of main memory among many processes. Virtual memory divides physical memory into chunks and allocates these chunks to different processes. Such a system must provide a protection mechanism to prevent one process to access memory of another process. Beyond sharing another motivation to use virtual memory is when a program requires more memory than available. In this case without virtual addresses the program itself has to care that the program did not access more memory than was available. Virtual memory automatically manages this by adding swapping mechanisms to secondary storage (usually hard disks). Another advantage is that program loading is simplified, because the program can be relocated anywhere in physical memory by just changing the mapping between virtual and physical addresses. The chunks in which the memory is divided can be of variable size (so called memory segments) or fixed size (so called memory pages). In the case of fixed size many processors support different page sizes, usually from 4KB to larger page sizes as 1MB, 2MB or 4MB. The translation process for the addresses is a combined software/hardware approach. The mapping and the address translation table is controlled by the operating system and located in main memory. But since every access to memory would require two main memory accesses, one to get the physical address and one to get the desired data, a caching mechanism for frequently used addresses is introduced. This cache is called translation lookaside buffer (TLB). The actual process of translation and the details of addressing is not covered here. What is of importance to the programmer is the influence of the page size. It is clear that a TLB miss is costly and must be avoided. Many aspects would favor larger page sizes:

- The page table is smaller.
- Swapping is more efficient.
- Memory can be mapped efficiently reducing TLB misses.

Against larger page sizes speaks, that they produce a larger memory fragmentation. Unfortunately different page sizes are often not equally well supported by the hardware. The advantage of e.g. large page sizes can be compensated by other effects. While other work shows large improvements with TLB aware optimization strategies [GG02] we found no serious limitations from TLB issues in our implementations. Switching to larger page sizes for an optimized Lattice Boltzmann implementation [Hau05] showed much lower TLB miss counts. But the overall performance increase was small.
Chapter 3

Instruction Set Architecture (ISA): The hardware software interface

It is recommended to read Appendix A as introduction to this chapter. Commonly most computers are following the von Neumann architecture, which describes the basic design for a digital computer. This architecture is also referred to as stored-program computer, a term which is used in the rest of this work to reflect, that also others apart von Neumann contributed to this design.

Key features of a stored-program computer are:

- Processors are controlled by instructions
- Instructions are represented as numbers
- Sequences of instructions are called programs and stored in memory

An ISA is a set of instructions a family of processors can execute. These instructions are the commands a given processor supports to control its behavior. The basic properties of an instruction set are dictated by practical thoughts. While there exist a variety of instruction set architectures they differ often only slightly. All computers are based on the same basic principle and design, resulting in instruction sets that are similar to each other. It is crucial to learn this language of the computer, because without a thorough knowledge it is not possible to understand performance and judge the quality of the code generated by the compiler. It is important to understand that the instruction code is the only interface to the processor and determines the performance of a program. In this Chapter the basic background is treated, for an in depth coverage of the topic we refer to common text books as e.g. \cite{HP05}. Experiments and practical examples with regard to the ISA influence on performance follow in Chapter 7. An introduction to the x86 and x86-64 ISAs is given in Appendix D and E. This Chapter is structured as follows: In Section 3.1 there is first given a short introduction to instruction set basics. In Section 3.2 we will try to figure out points in modern instruction sets influencing
performance. Section 3.3 describes the role and the problems compilers have with modern instructions sets.

### 3.1 Instruction Sets: A short introduction

The vocabulary to control a computer is called instruction set. A human readable text representation of machine language is assembly code. The common goal of a computer designer is to find a language that makes it easy to build the hardware and the compiler while maximizing performance and minimizing costs. It can be shown that there exists a set of instructions that is sufficient to control the execution of any sequence of operations. Every instruction set has to provide at least four groups of instructions:

- Arithmetic operation instructions
- Data transfer instructions
- Logical operation instruction
- Control flow instructions

Instruction sets can be categorized by different attributes. One categorization distinguishes an instruction set by the number of operands to instructions. This can vary from pure stack machines (zero operands) to instruction sets with a fixed number of three operands or a variable number of operands. With the rising importance of compilers new simple reduces instruction sets evolved, called RISC. Older ISAs, designed with hand written machine code in mind, were from there on referred to as CISC. RISC architectures try to simplify the instruction set in order to enable a simpler core logic in the processor and ideally suite automatic code generators. Typical properties of RISC instruction sets are (HP05):

- Uniform instruction format.
  - Refers to a uniform length of instructions and to an identical internal setup of instructions.
- Identical set of general purpose registers with commonly at least 16 registers.
- Simple addressing modes.
  - For example memory access only with dedicated load/store instructions.
- Few data types supported in hardware.

Nearly all newer ISAs obey above principles. And even the common x86 CISC instruction set converts its native instructions on the processor into RISC like instructions before execution.
Operands  Most instructions work on operands. To store operands of instructions there exists special storage in hardware called registers. These registers can be seen as something similar to variables in a high level language, with the difference that their number is limited. Their number differs from 8 in x86, 16 in x86-64, 32 in PowerPC to even 128 in IA64. Another important type of operand necessary to access memory is called memory operand. This operand is interpreted as an address which references a memory location. The last type of operands are immediates, which means constants that are hard coded into an instruction. To design an instruction set requires a balance between the number of instructions needed to execute a program, the number of clock cycles needed by an instruction and the possible speed of the clock. Hennessy/Patterson \[HP05\] suggest four design principles which help to get this balance:

1. Simplicity favors regularity: Keep all instructions the same size, fixed number of operands, keep the register fields at the same location.
2. Smaller is faster: More registers often means lower clock rate
3. Make the common case fast
4. Good design demands good compromises

Assembly language  Instruction code is a stream of numbers. This is difficult to read for human beings. Therefore there exists an intermediate clear text representation of the machine code called assembly. Assembly language is a symbolic language that can be directly translated into binary code. It consists of the machine instructions and assembly directives which influence the generation of the object code or executable format. This object code format is operation system and not directly instruction set related.

To be able to implement assembler code besides the knowledge of the instruction, the object file format, handled by assembler directives and the Application Binary Interface (ABI), regulating the code interaction on binary level, must be learned. It is of advantage to have a basic knowledge of the instruction set and assembler syntax in order to be able to read the code examples. An overview about the x86 ISA is given in Appendix D. An introduction to the gas (GNU assembler) assembler and assembler programming in general is found in Appendix F. The differences introduced by the x86-64 architecture are presented in Appendix E.

3.2 Modern ISAs: A clean design?

Instruction set architectures initially were invented to make code portable among different machines. The idea was, that a family of similar machines share the same instruction set. At that time, in the early sixties, every new machine had its dedicated instruction set and software written for one machine could only be run on this single machine type. Soon it
was realized that writing software is a complex and expensive effort. At that time nearly all software was implemented directly in machine language. To introduce instruction set architectures was a huge step forward for the importance and usability of computers in various fields. In general the situation of incompatible software was eased with a wider spreading of high level language compilers, which enabled a in principle hardware independent implementation of software. Still history showed that the concept of instruction set architectures created a certain momentum, meaning that once an instruction set architecture reached a widespread distribution, with a lot of software programmed for this architecture, it is difficult for new architectures to be introduced. This is also connected to the fact, that while software in principle can be implemented in an hardware independent fashion, it is still depending on an certain operating system. And writing an operating system for a new architecture is an expensive and complex task. All those factors caused, that today only few ISA have a significant position in the desktop and server computer area. The by far dominant ISA is the x86 family. This ISA originates in the early eighties. And despite many architectural changes it managed to maintain its dominant position.

At the time x86 was introduced the underlying micro architecture was completely different from today. Code density was important and hence the x86 architecture was a CISC instruction set. Moreover x86 initially was a 16 bit architecture, which means that the register width of the general purpose registers and therefore also the address width was 16 bit. Developments included the extension to 32 bit and later 64 bit, the introduction of a floating point co processor (FPU) and SIMD units. The underlying computer architecture of the hardware, implementing the instruction set, radically changed. To keep track it was necessary to introduce new instructions and instruction set extensions while retaining backward compatibility to old binary code. This introduced basic problems with regard to the performance of code. One problem is the non orthogonal nature of the x86 ISA. For one operation there exist many different ways and instructions to implement it. The best solution is highly hardware dependent and may vary even within the same family of processors. Because of the widespread use of the ISA it is a compromise to satisfy the demands of different hardware implementation concepts reasonably well. This fact can also be seen on the introduction of so called hint instructions, as the prefetch instructions in the multimedia extensions. Their behavior is by definition processor specific. One of the benefits of using an ISA the abstraction of the actual hardware within a computer family is with regard to performance is not existing anymore. Two main properties of modern ISA are responsible that many programs run with low efficiency and writing efficient instruction code is a complex and tedious task. First, the non orthogonal nature of many ISA introduces hardware dependent choices to be made. Second, modern ISA introduce hardware dependencies, hidden and also obvious, as with the multimedia extensions. Examples illustrating above statements can be found in Section 7.
3.3 From the high level language to the instruction level

Usually algorithms are implemented in a high level language adding another level of abstraction on top of the instruction set architecture. A language translator, the compiler, converts the high level language into assembler code which is then assembled into machine code. This adds major problems to the question of efficient code. As seen in the previous section there is a lot of hardware specific knowledge necessary to apply optimization techniques available in recent ISAs. Many architectural improvements also need adoptions in the instruction set to work properly. High level languages do not involve architecture specific information and do not adopt to hardware architectural changes. The compiler tries to extract enough information from the high level language code to apply the low level optimizations. This effort of the compiler to apply hardware specific optimizations is difficult, especially with a language as C, offering large degrees of freedom to the programmer. The first concern of the compiler is to produce correct code. Modern compilers try to enable a better optimization by offering compiler pragmas, with which the programmer can give additional information to the compiler and also often frees the compiler from the burden to ensure e.g. the absence of aliasing or conditions prohibiting vectorization of a loop. Also profile guided optimization is a technique used by modern compilers to improve their optimization effort. Especially the SIMD paradigm introduced in many ISAs is difficult to apply by compilers.

The performance of an implementation is decided on instruction code level. Modern ISAs introduce hardware dependencies and new low level programming paradigms, which are not reflected in the high level language. The compilers task is to bridge this gap with the knowledge exhibited by the high level programming language. Therefore there can be a significant performance loss relying on compiler generated instruction code. While there are many efforts to overcome this situation to fully utilize the performance potential of modern architectures it might be necessary to implement parts of the code directly in assembly code.
Chapter 4

Optimization Techniques

The basic optimization techniques used in this thesis as cache blocking, vectorization and data prefetching are well established. What is different in this thesis is the combination of those techniques and their application directly on the ISA layer. For an introduction to cache blocking techniques in general refer to the works [Wei01] and [Kow04]. A general introduction to optimization techniques as vectorization can be found in common text book on the topic as e.g. [GH01] and [Ger02]. A specific introduction to vectorization with multimedia extensions is e.g. [Bik04].

Section 4.2 introduces how runtime is created on computers. Because the scientific computing community is not in the comfortable situation to decide what hardware is build Section 4.1 introduces a pragmatic attitude to optimization on commodity processors. Section 4.3 gives annotations on optimization techniques investigated in previous works and comments on performance issues both related to modern architectures.

4.1 Basic Thoughts

Before thinking about optimization techniques it is necessary to understand the basic properties and optimization techniques used by a processor. The designers had certain situations in mind while designing the micro architecture of a given processor. As stated before a main benefit of the stored program computer is that it can perform various tasks. Of course these tasks may have very different demands to the hardware. As a consequence the final machine is a compromise to suite all relevant tasks best. Practically the hardware designer has a large set of benchmarks to find a good architectural compromise. Often the processor internally has different operation modes to allow task specific optimizations. There may be specific tasks, for which the processor works more efficiently than for others. This depends on which tasks in that large collection of benchmarks were regarded as more important to the designer group. For a good optimization it is very important to understand this fact: It is necessary to work in collaboration with the processor and understand what operation modes the hardware designer had in mind. The better a software fulfills the assumptions of the hardware the more efficient it works on a given processor. As worst case it might happen that a superior algorithm per-
forms so inefficient on a processor, that it is a better choice to use a worse algorithm, which runs more efficiently and might therefore at the end produce the result faster.

### 4.2 Runtime Considerations

Because runtime is the major motivation factor of an optimization effort in this thesis it is important to understand on an elementary level how runtime is created. The time a program needs to finish is computed as:

\[
time = \text{cycles} \times \text{clock rate}.
\]  

(4.1)

The cycles needed to finish a program can be seen as:

\[
cycles = \text{CPI} \times \text{number of instructions},
\]  

(4.2)

where Cycles per Instruction (CPI) is a measure how many cycles an instruction needs to complete in average. CPI characterizes therefore how efficiently a processor works within its capabilities. First the number of instructions is important and second how long each of those instructions takes to complete. For memory bandwidth limited algorithms, CPI is very high because each load needs to wait many cycles for completion. There are many other problems, which can cause the CPI to increase. Only a well balanced algorithm, which satisfies all architectural requirements best and only needs an acceptable number of instructions can lead to efficient code.

### 4.3 Performance Related Issues on Modern Architectures

The following remarks are valid for the class of algorithms covered in this thesis, which are iterative double floating point arithmetic algorithms acting on large spatial datasets. Modern processors are designed with two basic optimization strategies in mind:

- Streaming execution (Data Parallel) to enable a high data throughput with few operations which can be executed simultaneously. This execution pattern is similar to a vector like execution.

- Using a memory hierarchy to benefit from temporal and spatial data access locality.

Naturally this class of algorithms benefits from a streaming implementation. Large amounts of data is loaded and few operations, which can be executed simultaneously, are carried out. The result stream is then stored back to main memory. On the other hand the iterative nature the algorithms covered in this thesis favor a cache blocked implementation with small running inner loop bodies to exploit the property of available data reuse. The key insight for this class of algorithms is therefore to find a compromise to combine a streaming execution pattern, which is usually the best supported on modern processors, with a blocking implementation.
which exploits available data reuse and lowers the memory bandwidth requirements, examples for such a compromise can be found in the result chapters 9 and 10. There are two basic properties of a processor, which are important to judge how efficient a code runs on a given architecture: Arithmetic peak performance (in this case for double precision floating point calculations) and maximum sustained main memory bandwidth. The starting point is to answer two questions for a given architecture:

1. What has to be done to reach peak arithmetic performance?
2. What has to be done to reach peak memory bandwidth?

Despite there are common points valid for many architectures the solutions are architecture specific. During an optimization effort this knowledge has to be mapped to an algorithm to exploit the capabilities of a processor. The most important optimization at all is a data layout optimization. The effects of a proper data layout optimization are complex. They influence data access locality, may enable effective vectorization or lower the overall memory transfers. To find a good memory layout appropriate for a given architecture and algorithm is usually the most effective optimization step at all. Finally, for an iterative algorithm the memory bandwidth requirements can be reduced by reusing the data already in cache using blocking techniques.

4.3.1 Arithmetic Performance

The upper bound is the theoretical arithmetic peak performance. This upper bound has to be adopted with regard to the arithmetic characteristics of the code. The architectures covered in this thesis have to be distinguished into the x86 architectures and the IA64 architecture. The following remarks are only valid for the more common x86 processors. For IA64 related remarks refer to chapter 10. Every modern x86 processor has a special SIMD Arithmetic Logic Unit (ALU) and will only reach peak performance, if instructions executed on this unit are used. This group of instructions was already introduced in section 3.1 and is known under the term SSE instruction set extensions. The processors covered in this thesis can execute two floating point operations per cycle, this is only valid for an multiply add instruction mix. While the instructions allow to execute two operations at the same time, the architectures covered in this thesis do not use this opportunity and need two cycles per SSE instruction. Recent architectures, e.g. the Intel Core 2 architecture or the AMD Phenom processor, fully exploit the opportunities of the ISA executing most SSE instructions in one cycle. They can therefore execute 4 double precision floating point operations for a multiply add mix in each cycle. The upper bound is therefore determined by the instruction mix in the code. The classical benchmark which reaches sustainable peak performance is the triad routine (a detailed introduction of the triad routine is in Section 6.1). In the machine analysis phase an implementation of the triad should be found which reaches the highest performance. This
4.3.2 Data Layout Optimizations

As already stated data layout optimizations are usually the most effective optimization step at all. They improve the overall memory bandwidth requirements and influence many other issues as vectorization and inherent data reuse. It is difficult to give general hints because this optimization is specific to the algorithm and architecture and the best solution differs. General aims are:

- Lower the overall bus transfers.
- Enable more data reuse, best inherently.
- Enable long running inner loops over sequential data to favor efficient data prefetching.
- Enable aligned data access

In many cases a compromise is necessary to meet different demands and in rare cases it is necessary to decide for one of these issues to reach best performance. For concrete examples on memory layout optimizations refer to the result Sections 9.3 and 10.2.1.

4.3.3 Memory bandwidth

Many scientific algorithms are initially memory bandwidth limited. The calculations which can be carried out need so much data that the memory bus cannot deliver them in time. An useful metric in this context is the balance metric described e.g. in [Ber04]. It takes into account how balanced a given machine is in terms of the relation between peak floating point calculations to main memory bandwidth, meaning what performance a machine can sustain from main memory. This is set into relation to the algorithmic balance of an algorithm, which is the relation between how many flops the algorithm executes to how many double words it needs for that calculations. Because in the case of stencil based codes this metric neglects cache effects a more accurate prediction is achieved by the so called Loads per Miss metric also described in [Ber04]. By these metrics a first insight is achieved what fraction of the peak performance can be sustained from main memory on a given machine at best by a given algorithm. For a detailed analysis of the multigrid algorithm in terms of these metrics on different machines have a look in [Ber04]. The first question is how the sustainable main memory bandwidth of an algorithm can be improved and main memory bandwidth requirements be lowered. Here again the data layout chosen is of high importance, because
it governs in what order data is accessed and how much data reuse is implicitly exhibited. By a proper data access pattern, and that usually means a simple data access pattern, the compiler can apply memory access optimizations and the machine achieves a higher sustained memory bandwidth. An implicit data reuse lowers overall memory bandwidth requirements. An analysis for the x86 test machines can be found in Section 6.3.

**Data prefetching** If memory streams are deterministic an important optimization is to hide memory access latencies by introducing data prefetching. As explained in 2.2.2 this leads to a higher sustained memory bandwidth. Modern processors have hardware prefetch units to automatically detect uniform data access patterns and trigger data loads in advance. Additionally there exist special software prefetch instructions, which act as hints to the processor. How efficient these software prefetch instructions work depends on the actual implementation. The preferable solution is to use a data access pattern which is suited for the hardware prefetcher. This means ideally a long running sequential stride one data access with few total memory streams. Additionally the multimedia instruction set extensions offer dedicated prefetch instructions which enable better results for complex data access patterns, where the hardware prefetcher does not work well. On the test machines covered in this thesis the Intel processors showed no significant effect of software prefetch instructions for more complex data access patterns while the AMD architecture reached good results using software prefetch instructions (see also in Section 8.2).

**Lowering memory bandwidth requirements** If a code reaches a high arithmetic performance and uses the memory bus in an efficient way the final step to increase performance is to lower the number of overall memory bus transfers by exploiting available data reuse. The common technique to achieve that is cache blocking. Cache blocking tries to keep the data in cache while performing multiple subsequent operations on them and by that to minimize necessary main memory bus transfers. Therefore it is necessary to find a blocking variant that obeys the requirements for vectorization, which are data layout and alignment issues, and the requirements for data prefetching, which are a sequential data access pattern and long running inner loops. Because todays caches get larger this compromise is in most cases achievable. Depending on the algorithm it is often possible to go even a step further and try to prefetch data for data blocks that get updated in the near future in advance. An example of such an anticipatory advanced loading together with efficient vectorization and blocking techniques is presented in the results Chapter 10.2.1.
Chapter 5

Numerical Algorithms: Iterative Linear Solvers

5.1 Introduction

Many physical problems are described by mathematical models involving Partial Differential Equations (PDEs). While these models in principle fully describe a problem, relevant problems often cannot be solved analytically. Still it is possible to compute a numerical approximate solution for a problem, which is sufficient for many science and engineering applications. As the numerical parts are not the focus of this work the used numerical algorithms are introduced in short. For a more in depth theory we refer to common text books as e.g. [Hac93] or [Sch97]. The basic steps for a numerical solution of a PDE are:

1. Discretize the continuous problem
2. Solve the resulting algebraic system of equations

The most popular example for a boundary value problem is probably the Poisson equation. It is relevant in many areas as e.g. fluid dynamics, thermo- and molecular dynamics. A well known application of this model is e.g. the temperature distribution in an isotropic solid body (derived in [G]). All following derivations are for the 2D case, the 3D case is treated equivalent [Sch97]. The Poisson equation with homogeneous Dirichlet boundary conditions reads

\[ \Delta u(x, y) = -f(x, y) \quad \text{in } \Omega \]
\[ u(x, y) = 0 \quad \text{on } \partial \Omega \]

or written out:

\[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -f \] (5.1)

Numerical Discretization

The discretization of a PDE is a wide field of ongoing research. In the following one way to discretize the Poisson equation, which was used in this thesis is given.
For a numerical solution the PDE is approximately solved at a finite number of points. To do this the PDE is converted into a discrete form, meaning a form in which the partial derivatives are expressed in terms of discrete points. To choose the points in this thesis an equidistant structured grid is considered. The derivatives, in our case the Laplace operator, is discretized using the finite difference method, specifically a central differencing scheme which can be derived using a Taylor series expansion.

\[ (\Delta u)_{i,j} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\delta y^2} =: (\Delta^h u)_{i,j} \]  

(5.2)

The index \((i, j)\) denotes the function value at the grid point with the same indexes. To evaluate the value of the Laplacian of \(u\) at one point the value of \(u\) at that point, together with the values at its four neighbors are required.

This discretization process leads to a large, sparse linear equation system with one equation for every inner point. In matrix notation this linear system can be written as:

\[ Au^h = f^h \]  

(5.3)

This linear system of equation can be either solved using direct methods or approximately using iterative methods. A widespread iterative solver is the Gauss Seidel method [Har93], used throughout this thesis, and introduced in section 5.2. An improved form of iterative methods are multigrid methods. Multigrid methods use a property of iterative solvers: They act in a local scope using only neighboring points , and therefore can only smooth out high frequency errors visible in that scope. Low frequency errors, which involve many points need many iterations, because the information can only propagate slowly in each iteration. The multigrid method transforms the linear system to different scales and thereby ensures that the iterative smoother operates in its efficient scale. The specific multigrid method used in this thesis is described in section 5.3.

### 5.2 Gauss Seidel Method

It is not intended to introduce the Gauss Seidel method in a theoretical mathematical sense. Instead the actual algorithm is introduced to understand the optimizations applied later.
While the Gauss Seidel method is an iterative solver for Linear Equation System (LES) in the context of multigrid algorithms it is used as a smoother. Starting point is a LES which can be written as

\[ Ax = y \]  
(5.4)

The matrix \( A \) is disjoint into

\[ A = D - L - U \]  
(5.5)

where \( D \) are the values of the main diagonal, and \( L \) and \( U \) are the lower and upper triangular matrix. With substitution one obtains:

\[ (D - L) \cdot x = U \cdot x + y \]  
(5.6)

From above relation you can derive the following iterative method Hac96:

\[ D \cdot x^{k+1} - L \cdot x^{k+1} = U \cdot x^k + y \]  
(5.7)

where the \( k \) index denotes the iteration. Rewritten:

\[ d_{i,i} \cdot x_i^{k+1} - \sum_{j=1}^{i-1} l_{i,i} \cdot x_j^{k+1} = \sum_{j=i+1}^{i_{\text{max}}} u_{j,i} \cdot x_j^k + y_i \]  
(5.8)

which is equivalent to

\[ x_i^{k+1} = \frac{\sum_{j=1}^{i-1} a_{j,i} \cdot x_j^k + \sum_{j=i+1}^{i_{\text{max}}} -a_{j,i} \cdot x_j^k + y_i}{a_{i,i}} \]  
(5.9)

The Gauss Seidel method converges if the matrix \( A \) is positive definite or strictly diagonally dominant. The iterative solution can be described as: Starting from an initial guess \( x^0 \) the approximate solutions \( x^1 \) to \( x^n \) (\( n \) is the number of smoothing steps performed) are computed. Every iteration step begins with the computation of \( x_1^{k+1} \), because it only depends on values of \( x^k \). In the following \( x_j^{k+1} \) can be successively computed, depending of \( x_1^{k+1} \), up to the point one can compute \( x_i^{k+1} \). This method is efficient if the structure of the matrix \( A \) is sparse and the coefficients can be accessed by a fixed stencil as in a regular grid with e.g. a central difference discretization. This standard variant of the Gauss Seidel method is also called lexicographic Gauss Seidel. The ordering of the unknown vector with regard to the grid is illustrated in Figure 5.1 a. The resulting matrix structure for the discretized Poisson equation can be seen in figure 5.2.

A variant of the Gauss Seidel method is the red black Gauss Seidel method. Here the unknowns are reordered in the matrix and separated into red and black points in a checkerboard fashion on the grid. The ordering of the unknown vector with regard to the grid is illustrated Figure 5.1 b. The resulting matrix structure for the discretized Poisson equation can be seen in figure 5.3. This process splits every iteration into a red and a black half iteration. This has the advantage, that the updates in one half iteration are independent from each other and as a major advantage for the implementation, can be computed in arbitrary order or even parallel.
(a) Lexicographic ordering

(b) Red Black ordering

Figure 5.1: Numbering of the unknown vector

Figure 5.2: Coefficient matrix structure for Poisson equation.
In the following the algorithms for both methods is shown on our model problem the Poisson equation.

**Basic Algorithm of a 2D Gauss-Seidel implementation**

```plaintext
#allocate arrays of required size
alloc array U[0..m;0..n]
alloc array F[1..m-1;1..n-1]

#calculate discrete values on the grid for U and F
call init

#run for the desired number of iterations
repeat number of iterations times
    for j=1..(n-1)
        for i=1..(m-1)
            U[i;j] = ( U[i-1;j] + U[i+1;j] + U[i;j-1] + U[i;j+1] \
                        - dh*dh*F[i;j] ) / 4.0
        next i
    next j
end repeat
```

Figure 5.3: Coefficient matrix structure for Poisson equation in Red Black ordering.
5.3 Multigrid Method

As already mentioned iterative solvers have the property, that they can smooth out local short scale errors fast, but need many iterations for global long scale errors. The multigrid method, specifically the geometric multigrid method considered here, can overcome this weaknesses and has an optimal algorithmic complexity of $O(n)$. For a in depth introduction and more material on multigrid have a look at textbooks as [BHM00, TOS01].
5.3.1 Residual Equation

For the linear system

\[ A \cdot u = f \]  \hspace{1cm} (5.10)

let \( v \) be an approximate solution of the vector \( u \). Because \( u \) is unknown, instead of using a maximum- or euclidean norm of the error vector

\[ e = u - v \] \hspace{1cm} (5.11)

an equivalent norm of the residual is considered

\[ r = f - A \cdot v. \] \hspace{1cm} (5.12)

The residual is here only an indirect measure for the error – it describes point wise, how far the approximate solution satisfies the equation. Still the residual is a null vector, if no error is existent anymore. Through substitution you can derive the residual equation:

\[ r = A \cdot u - A \cdot v = A \cdot e \] \hspace{1cm} (5.13)

5.3.2 Basic Multigrid Scheme

Starting with an initial solution \( v \) for \( A \cdot v = f \) the high frequency errors are smoothed by a small number of iterations with an iterative solver. Then the belonging residual \( r = f - A \cdot v \) is computed. Using the residual equation it is now possible to find a good approximation of the error \( e \), in order to improve \( v \). To use the iterative smoothers in their efficient length scales, the computation of the approximate error \( A \cdot e = r \) is done on a coarser grid. In order to do that, it is necessary to compute a coarser representation of the residual \( r \) (restriction) and refine the computed coarse error for the following correction (interpolation). Because the solution gets corrected by an interpolated error often some post smoothing iterations are added.

The solution of the residual equation on a coarser grid can be applied recursively on subsequently coarser grids, until the grid size allows a fast exact solution of the problem. This procedure, subsequent coarsening of the residual and following refinement of the approximation of the error, is called V-cycle, and is applied repeatedly until a sufficient accuracy of the solution is reached.

As simple this basic algorithm is a lot of ongoing research on the various parts of it. Different ways of restriction and interpolation, variations of the smoother and different strategies of coarsening and refining can be applied depending on the underlying problem to be solved. For this thesis a simple model algorithm was chosen. The Poisson problem is discretized on a cubic, regular equidistant grid using a central differencing scheme. As smoother a red black Gauss Seidel smoother is considered. For the relaxation therefore the following seven point stencil is considered:

\[ V_{x,y,z} = \frac{1}{6} \cdot \left( V_{x-1,y,z} + V_{x+1,y,z} + V_{x,y-1,z} + V_{x,y+1,z} + V_{x,y,z-1} + V_{x,y,z+1} - dh \cdot F_{x,y,z} \right) \] \hspace{1cm} (5.14)
Points for which \(x + y + z\) is odd are red points. The refinement is computed by trilinear interpolation. For the restriction the weighted sum of the neighboring points is used, where the weights are chosen in opposite to the interpolation. The initial condition is always \(U_{x,y,z} = 0\) and the coarsening is up to one unknown.

**V-Cycle in pseudo-code**

```plaintext
function Vcycle(grid V, grid RS) {
    if (number_unknowns(V) == 1) {
        smooth(V,RS,1) /* exact solution */
    } else {
        new fine grid R
        new fine grid E
        new coarse grid Rcoarse
        new coarse grid Ecoarse

        smooth(V,RS,Pre)

        set_initialSolution(Esml)

        R = residual(V,RS)
        Rsml = restriction(R)

        Vcycle(Egrob, Rgrob)

        E = interpolate(Egrob)
        V = V + E

        smooth(V,RS,Post)
    }
}
```
Part II

Analysis
Chapter 6

Analysis of Performance Relevant Architectural Issues

This chapter investigates the relationship between instructions and their hardware implementation on the actual processor and the influences of the memory hierarchy on the performance. Entry point for optimization techniques are the optimization guides published by the processor manufacturers (e.g. [AMD07] and [Int02]). A thorough analysis of x86 micro architectures can be found in the paper [htt08a] published by A. Fog on his website. It is necessary to verify and analyze optimization techniques with micro benchmarks. Micro benchmark are small, isolated test cases. By measuring the runtime of these micro benchmarks different performance metrics as MFlops or MBytes/s can be derived in order to compare different implementations. The outcome of a machine analysis is what peak sustained performance can be reached and what has to be done to reach it. In Appendix B machine properties are listed in addition to below tests in an overview. The assignment to the test machines in Appendix B is realized through unique labels set in italic font.

6.1 Arithmetic Peak Performance

A common micro benchmark to reach arithmetic peak performance is the vector triad. In the form it is used here it has a mix of one multiply and one add operation with two load and one store stream. The covered processors have an optimized execution unit for an efficient fused multiply add operation. The arithmetic peak performance values of these processors are only valid for this special operation mix. An implementation of the triad benchmark in C is shown in code Listing 6.1.
A first test is a variant of above triad with only one load stream and without storing the result. It is intended to reach peak performance under the restriction that the L1 cache can sustain peak performance with one load stream. The test case is coded in x86 assembly (Listing 6.2). This code does not produce a correct result for the triad. The registers in which the result of the operation is stored are all different in order to reach peak performance.

Important to get high performance is to use the SSE SIMD instructions together with 16 byte loads. Besides this no further optimization is needed on instruction code level. No complicated software scheduling is necessary, in contrast simple code favors high performance. To reduce loop overhead, but more important to enlarge the basic block and thereby enable a more efficient instruction scheduling, unrolling by a factor of two to four has been shown to be sufficient on the covered architectures. Table 6.1 shows the results for the two reference architectures and a Core 2 processor, which was added to illustrate recent developments. The benchmark consists of a loop calculating 500 iterations with multiply add operations on a single vector. The efficiency of the processor can be seen at the CPI for this operation, an introduction to the CPI measure can be found in section 4.2. X86 processors decode the CISC instructions into smaller instructions which are suited to be executed on a pipelined ALU. Therefore it is sensible to apply the CPI measure on the already decoded instructions. The two reference processors have for double floating point calculations an ideal CPI of 0.5 for the multiply add operation. This means they can reach a minimum of 500 cycles for 1000 operations, which they nearly reach. The Core 2 in contrast can reach a CPI of 0.25 for these conditions. This means it needs minimum 250 cycles for 1000 operations. The reason, that the Core 2 reaches a smaller fraction of its theoretical peak performance can be the fact that the Pentium 4 tuned implementation was taken for it. As the Core 2 has a very high instruction throughput, L1 Cache access latencies might come into play and a register blocked implementation may be necessary to reach better results. As the Core 2 is not in the focus of this work, this was not investigated further.
ANALYSIS OF PERFORMANCE RELEVANT ARCHITECTURAL ISSUES

Table 6.1: Arithmetic peak performance

<table>
<thead>
<tr>
<th>Processor</th>
<th>peak performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 4</td>
<td>6.01 GFlops (520 cycles)</td>
</tr>
<tr>
<td>Athlon 64</td>
<td>4.32 GFlops (545 cycles)</td>
</tr>
<tr>
<td>Core 2</td>
<td>9.27 GFlops (315 cycles)</td>
</tr>
</tbody>
</table>

Listing 6.2 shows the loop body for the Pentium 4 tuned peak flop benchmark.

Listing 6.2: Peak performance benchmark for the Pentium 4

```asm
1  movapd xmm1, [esi + eax*8]
2  addpd xmm6, xmm7
3  mulpd xmm1, xmm7
4  movapd xmm2, [esi + eax*8 + 16]
5  addpd xmm5, xmm7
6  mulpd xmm2, xmm7
7  movapd xmm3, [esi + eax*8 + 32]
8  addpd xmm0, xmm7
9  mulpd xmm3, xmm7
10 movapd xmm4, [esi + eax*8 + 48]
11 addpd xmm2, xmm7
12 mulpd xmm4, xmm7
```

6.2 In Cache Peak Performance

More relevant for practical applications is what fraction of the peak performance can be sustained from the caches. The triad benchmark as shown in 6.1 is taken as test code, The results are listed in table 6.2 for the L1 cache and table 6.3 for the L2 cache. The first measured test case is the triad with one load stream. In difference to the test case in the previous section now the correct triad is taken into account. The second load stream is added and finally the full triad with two load and one store streams is measured. For one load stream the L1 cache can nearly sustain peak performance on all machines as shown in the previous section. While Pentium 4 and Athlon 64 are already cache bandwidth limited for the full triad the Core 2 can sustain its performance from L1 cache for this case. Pentium 4 and Athlon 64 are despite the higher theoretical peak performance of the Pentium 4 on the same level due to the comparable L1 bandwidth. It has to be be mentioned, that two different assembly implementations were used: One plain RISC like implementation and one optimized version in which load-execute instructions are used for the Athlon 64, more information on this issue can be found in section 7.2. From the L2 cache all architectures are cache bandwidth limited. The Athlon 64 has compared to the other architectures a low L2 cache bandwidth.
6.3 Sustainable Main Memory Bandwidth

In common programming languages the sustained main memory bandwidth is influenced mainly by the order in which data is accessed. The instruction set level offers many possible optimizations to improve the sustainable main memory performance. In the following these optimizations are explained in more detail. To illustrate each optimization the improvement is shown on the example of memcpy, an elementary memory bound operation, which copies data from one to a different memory location. For comparison the achieved bandwidth of the system memcpy call is taken as base (Table 6.4). The executable is the same 32 bit binary for all benchmark runs. It only differs in the assembler prefetch code, which is tuned for each platform. The benchmark copies a 128 MB large memory buffer. Besides the system memcpy, result Table 6.4 shows the result for a standard assembler version using the general purpose registers for copying (Listing 7.2). The following section concentrates on the best results, which could be reached on the test machines. The implications of the ISA influencing memory bandwidth is treated in Section 7.1. In this Section also the complete code listings can be found. More results of memory bandwidth optimization on the example of a vector triad operation can be found in [HZTW05].

6.3.1 16 Byte Loads

The smallest data unit for transfers inside the memory hierarchy is a cache line. From L1 cache to registers it is word length (4 byte on x86). The SSE extension adds 16 byte wide registers, for which special 16 byte load instructions exist. These load instructions reduce the
number of total loads and enable a higher memory load bandwidth (Table 6.4, Listing 7.3).

### 6.3.2 Non Temporal Stores

Main memory works most efficient by operating in burst mode, see also in section 2.2.2. Modern architectures have special caches to combine several sequential stores, usually of cache line length, and burst these chunks at once to main memory. Moreover it is favorable for a streaming execution pattern to bypass the cache hierarchy for stores, which contain data not reused in the near future also preventing cache pollution and the read for ownership on a store miss.

Modern architectures have special capabilities to support this process by dedicated store instructions. These instructions add a non temporal hint to stores. This non temporal hint tells the processor that the store can be written to main memory without ensuring data consistency. Because of the hint character of this instruction the hardware implementation can use this information, but does not need to use it. On most implementations these non temporal stores dramatically affect sustainable store main memory performance as can be seen in Table 6.5 (Listing 7.4). The first column shows the SSE version with temporal stores, the second column uses non-temporal stores.

### 6.3.3 Data Prefetching

Data prefetching is an optimization used by all modern processors. While the hardware prefetch units are supposed to work automatically the multimedia instruction set extensions offer dedicated prefetch instructions which shall bring improvements for complex data access patterns, where the hardware prefetcher does not work well. As the non temporal stores these instructions are only hint instructions, which means the processor can ignore them. On our test machines the Intel processors showed no significant effect of software prefetch instructions for more complex data access patterns. For comparison a data load benchmark was implemented, with one stride one load memory stream (Table 6.6, Listing 7.6). The first column shows the basic performance without any prefetching. In the second column one cacheline wise move into a general purpose register triggers the move of data into the cache hierarchy. In the third column the same was done using a software prefetch instruction. It can be seen, that for this simple case the software prefetch instruction shows equal or slightly better performance than the cacheline wise move.

Still if applied on the memcpy with an additional memory store stream the software prefetch

<table>
<thead>
<tr>
<th>Processor</th>
<th>memcpy (system)</th>
<th>memcpy (ASM 4 byte)</th>
<th>memcpy (SSE 16 byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 4</td>
<td>2303 MB/s</td>
<td>2476 MB/s</td>
<td>2556 MB/s</td>
</tr>
<tr>
<td>Athlon 64</td>
<td>2740 MB/s</td>
<td>2735 MB/s</td>
<td>2914 MB/s</td>
</tr>
<tr>
<td>Core 2</td>
<td>3417 MB/s</td>
<td>3036 MB/s</td>
<td>3128 MB/s</td>
</tr>
</tbody>
</table>

Table 6.4: Memcpy: Baseline
Table 6.5: Memcpy: Non temporal stores

<table>
<thead>
<tr>
<th>Processor</th>
<th>memcpy (SSE)</th>
<th>memcpy (SSE NT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 4</td>
<td>2556 MB/s</td>
<td>3889 MB/s</td>
</tr>
<tr>
<td>Athlon 64</td>
<td>2914 MB/s</td>
<td>4436 MB/s</td>
</tr>
<tr>
<td>Core 2</td>
<td>3128 MB/s</td>
<td>4787 MB/s</td>
</tr>
</tbody>
</table>

instruction shows no significant effect on the Intel architectures, while on the Athlon 64 it still works and causes a significant improvement in main memory bandwidth (Table 6.7 column 2).

### 6.3.4 Block Prefetching

The memory controller requires a significant latency to switch between loads and stores. The company SGI published an article on the optimization of memory operations for the Intel Pentium processor ([SGI96](#)), disappeared from the SGI Website in 2003), in which many of the techniques presented here were already proposed. In this paper a technique referred to as block prefetching is presented. The data is loaded block wise into cache with a separate prefetch loop. The prefetch loop executes one move into a general purpose register per cacheline and load stream. This cache block is then stored back in a separate loop. For further results of the block prefetch technique have a look in Section 8.2.

Listing 6.3: Algorithm Block Prefetching for vector triad

```plaintext
for SIZE
    for BLOCKSIZE
        preload A
    endfor
    for BLOCKSIZE
        preload B
    endfor
    for BLOCKSIZE
        computations
        store stream
    endfor
endfor
```

Both the Pentium 4 and Athlon 64 reach their best performance using this block prefetch technique (Table 6.7 column 3, Listing 7.7). In contrast the Core 2 architecture already

Table 6.6: Memread: Software Prefetching

<table>
<thead>
<tr>
<th>Processor</th>
<th>memread [RISC]</th>
<th>memread (CL touch)</th>
<th>memread [SSE SWNTA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 4</td>
<td>4101 MB/s</td>
<td>5697 MB/s</td>
<td>5698 MB/s</td>
</tr>
<tr>
<td>Athlon 64</td>
<td>3179 MB/s</td>
<td>5914 MB/s</td>
<td>6094 MB/s</td>
</tr>
<tr>
<td>Core 2</td>
<td>4949 MB/s</td>
<td>5805 MB/s</td>
<td>6080 MB/s</td>
</tr>
</tbody>
</table>
Table 6.7: Memcpy: Software Prefetching

<table>
<thead>
<tr>
<th>Processor</th>
<th>memcpy (SSE NT)</th>
<th>memcpy (SSE NT SWPF)</th>
<th>memcpy (Block Prefetch)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 4</td>
<td>3889 MB/s</td>
<td>3952 MB/s</td>
<td>4644 MB/s</td>
</tr>
<tr>
<td>Athlon 64</td>
<td>4436 MB/s</td>
<td>5372 MB/s</td>
<td>5843 MB/s</td>
</tr>
<tr>
<td>Core 2</td>
<td>4787 MB/s</td>
<td>4765 MB/s</td>
<td>4916 MB/s</td>
</tr>
</tbody>
</table>

shows good performance with the standard non-temporal version and does not improve much with the block prefetching version. It has to be mentioned that the Core 2 has a much higher theoretical peak memory bandwidth (10.8 GB/s) compared to the other architectures (6.4 GB/s). Related to the theoretical peak main memory bandwidth its sustained peak performance is disappointing.

6.4 Influence of the TLB on Performance

An introduction to virtual addressing and the use of a TLB is in Section 2.2.3. The main argument for a smaller page size is to avoid additional memory consumption due to memory fragmentation. Previous research in the DiME project [Kow04] experienced a low performance for cache blocked algorithms on modern architectures as e.g. the Pentium 4 with three dimensional data sets. It was not possible to figure out what was exactly the reason for this fact. One suspect was that TLB misses caused the problems with a large number of memory pages, that are in use at the same time. This was also motivated by the small TLB on the Pentium 4 with initially only 64 entries with 4 kB page size and a high number of TLB cache misses measured with hardware performance counters. The number of TLB misses can be reduced by using larger page sizes, e.g. 1MB. Most processors allow to switch page size. Tests with huge pages [Hau05] showed a lower performance with a very low memory bandwidth on the Pentium 4. The measured performance counter TLB misses were significantly reduced with large pages. The very low load memory bandwidth may indicate that the hardware prefetch unit is not working with large pages support enabled. The experiments were stopped at this stage. To figure out the impact of the TLB tests with an increasing number of memory streams where done. Memory bandwidth decreases with every additional stream, but no indication was found that this is alone caused by the TLB. Exactly the same phenomena was found on the Athlon 64 with a much larger TLB. TLB influences were not investigated further because other techniques as vectorization and prefetching resulted in a much improved performance compared to the old DiME codes. Still it is not finally proved that TLB issues do not play a limiting role. Interesting in this context is a paper by [GG02] describing a TLB aware implementation of a matrix matrix multiplication. For more details on the TLB tests refer to [Hau05].
6.5 Summary

The result of this chapter is that it is possible to reach over 80% of arithmetic and memory bandwidth peak performance on the reference architectures. In the case of arithmetic peak performance this is achieved by using the multimedia instruction set extensions. For memory bandwidth it is not sufficient to use the ISA in a straightforward way. For the memory copy operation the block prefetching algorithm together with cacheline wise loads and not the use of the dedicated prefetch instructions showed the best results.
Chapter 7

Examples on the Influence of the ISA on performance

In the following sections examples are given to illustrate the influence of ISA related issues on performance. In section 7.1 the influence of different software implementations of the memory copy operation on performance are shown. Section 7.2 investigates the impact of different addressing modes on L1 cache bandwidth on the Athlon 64. An in depth analysis of ISA and assembler related issues can be found in [htt08c].

7.1 Choosing the Right Instructions on the Example of Mempcy

Memory copy is the elementary form of a memory bound operation. A stream of subsequent data chunks is loaded into register and written back to memory to a different location. To illustrate the influence of different instructions a series of memcpy versions, copying a 128 MB data block, was implemented for the Pentium 4 processor. The base version is the system memcpy provided by the standard C library. The most compact version with regard to code size is the CISC version using the string instructions of the x86 ISA:

Listing 7.1: memcpy implemented in assembler using CISC instructions

```
1 mov edi, [ebp+8]
2 mov esi, [ebp+12]
3 mov ecx, [ebp+16]
4 .align 16
5 rep movsb
```

CISC instructions are often replaced on modern processors with a series of smaller RISC like instructions. A RISC version is the following:

Listing 7.2: memcpy implemented in assembler using RISC instructions

```
1 mov edi, [ebp+12]
2 mov esi, [ebp+8]
```
Above version also implements four times unrolling to reduce loop overhead and enable better pipelining. Up to now general purpose registers are used. The next software implementation variant uses the SSE registers available on every modern x86 processor. A shortened example is:

Listing 7.3: memcpy implemented in assembler using SSE instructions

As the loop preparation code is the same as before from here on only the loop body is shown. The SSE instruction set extensions provide a wide range of move instructions for different data types. A main benefit of using SSE are the 128 bit wide xmm registers. Less instructions are needed for the same task. As memcpy does not have any temporal locality it makes sense for the processor to bypass the caches. The Pentium 4, and most other modern processors, have so called write combine buffers. These buffers collect writes to subsequent addresses and directly burst the data e.g. 64 byte wise to memory. The SSE ISA has instructions
to enable the processor to use this optimization. This group of instructions is called non-temporal store instructions. Replacing the `movaps` with the `movntps` instruction enables this optimization:

Listing 7.4: memcpy implemented in assembler using SSE instructions and non temporal stores

```asm
1:  movaps  xmm0, [edi+ecx*4+0]
2      movntps [esi+ecx*4+0], xmm0
3      movaps  xmm1, [edi+ecx*4+16]
4      movntps [esi+ecx*4+16], xmm1
5      movaps  xmm2, [edi+ecx*4+32]
6      movntps [esi+ecx*4+32], xmm2
7      movaps  xmm3, [edi+ecx*4+48]
8      movntps [esi+ecx*4+48], xmm3
9      add  ecx, 16
10     cmp  ecx, edx
11     jb   lb
```

To illustrate the fact that different instructions with the same purpose show a similar performance a SSE2 version of above memcpy was implemented:

Listing 7.5: memcpy implemented in assembler using SSE2 instructions and non temporal stores

```asm
1:  movapd  xmm0, [edi+ecx*4+0]
2      movntpd [esi+ecx*4+0], xmm0
3      movapd  xmm1, [edi+ecx*4+16]
4      movntpd [esi+ecx*4+16], xmm1
5      movapd  xmm2, [edi+ecx*4+32]
6      movntpd [esi+ecx*4+32], xmm2
7      movapd  xmm3, [edi+ecx*4+48]
8      movntpd [esi+ecx*4+48], xmm3
9      add  ecx, 16
10     cmp  ecx, edx
11     jb   lb
```

The `pd` after each instruction stands for packed double instructions, meaning that each instruction operates on two double values at the same time. For memcpy it is of no importance what is inside the register. The `ps` suffix above stands for packed single with 4 single values in one register. The ISA states that one instruction operates on two values at the same time. The ISA would allow to execute these two operations independent from each other. The Pentium 4 does not exploit this opportunity and needs two cycles for one SSE instruction. This means it can operate on one double or two single values in one cycle.

Its successor the Intel Core 2 architecture actually can execute one SSE instruction in each cycle. There is one optimization left the ISA provides for our purpose: Software prefetch instructions. Every modern processor has a data prefetch unit, which operates automatically.
For complicated cases it is difficult for the hardware prefetcher to recognize more complicated data access patterns. For that purpose software prefetch instructions were added to support the prefetcher. The ISA marks these instructions as hint instructions. The processor does not need to execute these instructions. The memcpy routine has a very simple data access pattern with one stride one data stream.

Listing 7.6: memcpy implemented in assembler using SSE2 instructions and non temporal stores with prefetch instructions

```
1:  
2    prefetchnta [edi+ecx*8+1024]  
3    movapd xmm0, [edi+ecx*4+0]  
4    movntpd [esi+ecx*4+0], xmm0  
5    movapd xmm1, [edi+ecx*4+16]  
6    movntpd [esi+ecx*4+16], xmm1  
7    movapd xmm2, [edi+ecx*4+32]  
8    movntpd [esi+ecx*4+32], xmm2  
9    movapd xmm3, [edi+ecx*4+48]  
10   movntpd [esi+ecx*4+48], xmm3  
11   add ecx, 16  
12   cmp ecx, edx  
13   jb 1b  
```

Up to now optimization techniques are used, which the ISA provides for this case. As can be seen above there is a variety of possible ways offered by the ISA to implement a simple operation as memcpy. Moreover the hardware implementation has the freedom to implement the ISA in different ways, making it difficult for the compiler to choose the best way for a software implementation. To further optimize the memcpy a deeper knowledge about the memory subsystem is necessary. An important fact is that there is a latency to switch from memory load to store access. Therefore it is favorable to separate stores from loads. To bring data into cache explicitly an effective technique are cacheline wise loads into register. If one 4 byte load into an general purpose register is executed the whole 64 byte cacheline is loaded into cache. This technique in contrast to prefetch instructions offered by SSE always works, because the processor has to execute these instructions. To distinguish this technique from software prefetch instructions it will be referred to as preloading from now on. These two optimizations together with the use of non-temporal stores and the usage of the SSE registers result in the final version of memcpy (Listing 7.7).

Listing 7.7: memcpy implemented in assembler using SSE2 instructions and non temporal stores with prefetch instructions and usage of preloading

```
1 .mainloop:  
2   xor eax, eax  
3   .align 16  
4   1:  
5   mov ebx, [edi + eax *4 + 0]  
```
A summary of the results can be seen in Table 7.1. It can be said, that the Athlon 64 shows an overall better main memory bandwidth on the memory copy operation. The improvements are comparable with the exception of going from the CISC to the RISC version and the use of software prefetch instructions. The difference of the CISC and RISC version may be caused by the design of the front end on both processors. The comparable results of both versions on the Pentium 4 suggests that the CISC instruction is replaced with a series of RISC microinstructions. In the optimization manual for the Athlon 64 [AMD07] it is recommended to not use the rep instruction due to high latencies. The result confirms this with a significantly lower performance of the CISC version compared to the RISC version. The use of software prefetch instructions results in a large performance increase on the Athlon 64. On the Pentium 4 in contrast the increase is small. This confirms the experience, that on the Pentium 4 the software prefetch instructions have often no effect. On both reference architectures the version using the block prefetching technique is the best performing, reaching 6 GB/s on the Athlon 64. The lower 4.6 GB/s on the Pentium 4 is due to the lower store performance.
Table 7.1: Overview memcpy results

<table>
<thead>
<tr>
<th></th>
<th>Pentium 4</th>
<th>Athlon64</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td>2303 MB/s</td>
<td>2740 MB/s</td>
</tr>
<tr>
<td>CISC</td>
<td>2481 MB/s</td>
<td>2001 MB/s</td>
</tr>
<tr>
<td>RISC</td>
<td>2424 MB/s</td>
<td>2834 MB/s</td>
</tr>
<tr>
<td>SSE</td>
<td>2555 MB/s</td>
<td>2914 MB/s</td>
</tr>
<tr>
<td>SSE Non Temporal</td>
<td>3923 MB/s</td>
<td>4436 MB/s</td>
</tr>
<tr>
<td>SSE2 Non Temporal</td>
<td>3928 MB/s</td>
<td>4426 MB/s</td>
</tr>
<tr>
<td>SSE NT Software Prefetch</td>
<td>4012 MB/s</td>
<td>5206 MB/s</td>
</tr>
<tr>
<td>SSE2 Block Prefetch</td>
<td>4644 MB/s</td>
<td>6030 MB/s</td>
</tr>
</tbody>
</table>

7.2 Addressing Mode Impact on the Athlon 64 Architecture

The Athlon 64 in our tests initially reached a disappointing L1 load cache bandwidth. In the following example an assembler routine is taken, which adds two vectors which fit in the L1 Cache, to investigate what influences L1 cache bandwidth on the Athlon 64. To see how well the L1 cache bandwidth scales first a vector is added up with a scalar, which is hold in register. Because the two reference architectures can execute one add per cycle this test codes peak performance is 50% of the multiply add peak. This add peak performance is nearly reached as can be seen in Table 7.2. L1 cache bandwidth is no limiting bottleneck here. For comparison the peak performance reachable with multiply add code is shown (Table 7.2 MultAdd).

Now a second vector is added instead of the scalar, the bandwidth requirements effectively double (Table 7.2 Add 2). On both processors the arithmetic peak performance is now L1 cache bandwidth limited. Still the Pentium 4 can sustain 36% of its theoretical peak performance reaching a cache bandwidth of 37.4 GB/s while the Athlon 64 drops to 24% with a L1 bandwidth of 19 GB/s. A value the Pentium 4 is able to sustain from L2 cache. Code listing 7.9 shows the addressing used.

Listing 7.8: RISC data load

1. movdqa xmm1, [x+ecx*8]
2. movdqa xmm3, [y+ecx*8]
3. addpd xmm3, xmm1

This is RISC like code with loading the data first into register and then perform the add in a separate instruction. x86 allows a different addressing with the possibility to add memory references in nearly every instruction. An alternative way to write this operation is seen in Listing 7.9.

Listing 7.9: CISC data load

1. movdqa xmm4, [x+ecx*8]
2. addpd xmm4, [y+ecx*8]
This alternative addressing is used in the version Add 2 var in Table 7.2. The Athlon 64 reaches with 33.3 GB/s nearly two times the bandwidth than before with a RISC-like addressing. The Intel Pentium 4 improves only slightly. The reason is that the Athlon 64 architecture is optimized for load-execute instructions. Most of these instructions are so called direct path decodable, which means that these instructions can be decoded into macro ops very efficiently [AMD07]. The front end can decode up to three direct path instructions in one cycle enabling good opportunities for the scheduler to exploit ILP. If these load-execute instructions are split up decoding bandwidth is wasted and pressure on the registers is increased causing scheduler stalls because of processor front end decoding limitations. The Pentium 4 on the other hand can only decode one instruction per cycle. For small repeating code blocks this is no limitation because the code stream can be served from the trace cache, a small instruction cache holding the already decoded micro op instructions. This is an example of a hardware dependent ISA issue influencing performance dramatically.

<table>
<thead>
<tr>
<th></th>
<th>Pentium 4</th>
<th>Athlon64</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Add</strong></td>
<td>2920 MFlops (44.6 %)</td>
<td>2338 MFlops (48.7 %)</td>
</tr>
<tr>
<td></td>
<td>23.3 GByte/s</td>
<td>18.7 GByte/s</td>
</tr>
<tr>
<td><strong>MultAdd</strong></td>
<td>5229 MFlops (81 %)</td>
<td>4153 MFlops (86 %)</td>
</tr>
<tr>
<td></td>
<td>20.9 GByte/s</td>
<td>16.1 GByte/s</td>
</tr>
<tr>
<td><strong>Add 2</strong></td>
<td>2339 MFlops (36 %)</td>
<td>1187 (24 %)</td>
</tr>
<tr>
<td></td>
<td>37.4 GByte/s</td>
<td>19.0 GByte/s</td>
</tr>
<tr>
<td><strong>Add 2 var</strong></td>
<td>2454 MFlops (38 %)</td>
<td>2082 MFlops (43.3 %)</td>
</tr>
<tr>
<td></td>
<td>39.2 GByte/s</td>
<td>33.3 GByte/s</td>
</tr>
</tbody>
</table>
Chapter 8

Examples on Compiler Related Performance Issues

In this Chapter it is illustrated on two examples, what influence the compiler has on an optimization effort. While throughout the work the influence of the compiler is not taken into account, these examples shall support the statement, that the compiler does not reliably generate efficient code. In Section 8.1 the compiler generated code for a nested loop structure typical for optimized scientific codes is analyzed. After that in Section 8.2 the possibilities of improving the compiled code quality by using pragmas and intrinsics are investigated. A in depth analysis of compiler and programming language related performance issues can be found in the paper [htt08b].

8.1 Loop Overhead

A primary task of a compiler is register scheduling and efficient addressing. Especially cache optimized code with its nested loop structures and multidimensional array addressing is a demanding task in this context. The small number of general purpose registers on the x86 architecture makes this task even more difficult. To illustrate the differences among compiler generated codes an example written in Fortran 77 was taken from the DiME 2D demonstration code. Except for the loop indexing it consists solely of floating point instructions in the high level code, hence all integer instructions in the instruction code can be seen as overhead. The most critical part of the code are the innermost loops. These sections should contain as few integer instructions as possible, because a large amount of the total execution time is spent on these code blocks. To analyze the instruction mix a Perl script which parses the loop structures and prints out the instruction mix for each code block was implemented. The analyzed Fortran code block is the following nested loop structure:

Listing 8.1: DiME blocked loop construct (Fortran 77)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>do i=m*2,n-1,2</td>
</tr>
<tr>
<td>2</td>
<td>do j=1+mod(i+1,2),n-2-1,4</td>
</tr>
</tbody>
</table>
The code was compiled with optimization flags for the Intel Pentium 4 Prescott processor. The analyzed compilers are listed in Table 8.1 (F stands for floating point instructions, I for all other instructions). The compiler flags used can be seen in Table 8.2.

The GNU compilers g77 3.4.5 and gfortran 4.0.0 need a large number of integer instructions for addressing the arrays in the inner most loops. Even worse they need multiple stack accesses, 74 for the gfortran 4.0 against 0 for the Intel compilers or newer version of the gfortran compiler. Integer instructions are cheap in relation to floating point calculations on modern processors. Still this large instruction overhead degrades performance, especially together with the stack accesses in the innermost loop. For two compilers the cycles taken to execute above code snippet were measured. Comparing the Intel ifort 8.0 result with the GNU gfortran 4.0 with regard to cycles taken for that code snippets shows $120 \times 10^6$ versus $303 \times 10^6$ cycles. The gfortran compiler improved with versions 4.1.0 and 4.2.3 and reaches the quality of the Intel compilers for our code example as can be seen in table 8.1. The Intel compilers and latest gfortran compilers keep the addressing of the arrays completely outside of the innermost loop. The results of this static code analysis show, that it cannot be taken for granted that the compiler reliably generates efficient code. No compiler was able to apply efficient instructions as packed arithmetic instructions and 16 byte moves in the innermost loop. The usage of compiler pragmas !$\text{pragma ivdep}$ and !$\text{pragma vector aligned}$ enabling a more

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Instr. count</th>
<th>Overall ratio [F/I]</th>
<th>Total/Int/Float/Stack</th>
<th>Loop ratio[F/I]</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU g77 3.4.5</td>
<td>456</td>
<td>0.89</td>
<td>302/116/186/97</td>
<td>1.60</td>
</tr>
<tr>
<td>GNU gfortran 4.0.0</td>
<td>475</td>
<td>0.88</td>
<td>292/106/186/74</td>
<td>1.75</td>
</tr>
<tr>
<td>GNU gfortran 4.1.0</td>
<td>386</td>
<td>1.33</td>
<td>217/31/186/6</td>
<td>6.0</td>
</tr>
<tr>
<td>GNU gfortran 4.2.3</td>
<td>432</td>
<td>1.06</td>
<td>191/5/186/0</td>
<td>37.2</td>
</tr>
<tr>
<td>Intel ifort 8.0</td>
<td>208</td>
<td>1.63</td>
<td>70/8/62/0</td>
<td>7.75</td>
</tr>
<tr>
<td>Intel ifort 9.0</td>
<td>344</td>
<td>3.06</td>
<td>192/2/190/0</td>
<td>95</td>
</tr>
<tr>
<td>Intel ifort 10.1</td>
<td>333</td>
<td>3.12</td>
<td>193/3/190/0</td>
<td>63</td>
</tr>
</tbody>
</table>
aggressive vectorization had no effect in the generated assembler code.

8.2 Software data prefetching

The vector triad is taken as test case. It is investigated, what can be done to support the processor in its effort to generate efficient code. Efficient code means for this case code reaching a high sustained main memory bandwidth. These tests are using the Intel icc compiler (8.1). The C code for the test case can be seen in Listing 8.2.

```
for (i = 0; i< SIZE; i++){
  A[i] = B[i] + alpha * C[i];
}
```

Alpha is a double scalar variable. A, B, C are double arrays, the vector size is such that the triad is main memory bound. The first way to improve the code generation is to supply additional informations to the compiler. For this purpose there exists a variety of compiler directives (in the form of compiler pragmas) which can be added to support the optimization effort of the compiler. An improved version including compiler pragmas shown on Listing 8.3.

```
#pragma vector nontemporal
#pragma vector aligned
#pragma prefetch B C

for (i = 0; i< SIZE; i++){
  A[i] = B[i] + alpha * C[i];
}
```

These pragmas tell the compiler that the store vector is non temporal, that the vectors are 16 byte aligned and that it is desired to prefetch the vectors B and C. Using the flags 
- S -fast -ansi -c99 The compiler generates the following code:

Table 8.2: Compiler Flags

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel ifort</td>
<td>-O3 -xP -no-alias</td>
</tr>
<tr>
<td>GNU g77 3.4.5</td>
<td>-O3 -march=pentium4 -ffast-math -fomit-frame-pointer</td>
</tr>
<tr>
<td></td>
<td>-funroll-loops -fargument-noalias-global -msse2 -mfpmath=sse</td>
</tr>
<tr>
<td>GNU gfortran 4.0.0</td>
<td>-O3 -march=prescott -ffast-math -floop-optimize2</td>
</tr>
<tr>
<td></td>
<td>-funroll-loops -fargument-noalias-global -msse2 -mfpmath=sse</td>
</tr>
<tr>
<td>GNU gfortran 4.1.0</td>
<td>-O3 -march=prescott -ffast-math -floop-optimize2</td>
</tr>
<tr>
<td></td>
<td>-funroll-loops -fargument-noalias-global -msse2 -mfpmath=sse</td>
</tr>
<tr>
<td>GNU gfortran 4.2.3</td>
<td>-O3 -march=prescott -ffast-math -floop-optimize2</td>
</tr>
<tr>
<td></td>
<td>-funroll-loops -fargument-noalias-global -msse2 -mfpmath=sse</td>
</tr>
</tbody>
</table>
The code looks similar to a version a human being would write. It does slight instruction scheduling, uses non temporal stores and vectorizes the code. The unroll factor is four. This also makes sense because the *Pentium 4* uses a small trace cache, which only works properly if the loop body is not too big. But it does not prefetch the two vectors, which is sensible because in such a simple case the hardware prefetch unit should be able to detect the data access pattern. For cases it is desired to force the compiler to generate software prefetch code compiler intrinsics can be used. Above code changes as follows:

Listing 8.5: Applying compiler intrinsics

```
#pragma vector nontemporal
#pragma vector aligned
#pragma prefetch B C
for (i=0;i<SIZE; i++){
    _mm_prefetch((const char*) B+i*8+256, _MM_HINT_T1);
    _mm_prefetch((const char*) C+i*8+256, _MM_HINT_T1);
    A[i] = B[i] + alpha * C[i];
}
```

Two prefetch intrinsics were added, one for each load vector. The generated code now includes at the beginning of the loop body:

- `prefetcht1 256(%ecx,%eax,8)`
- `prefetcht1 256(%edx,%eax,8)`

The vector triad is a simple example, the compiler can also optimize it properly without any pragmas. Table 8.3 shows the achieved performance.
Still the effect of the pragmas can be seen in the code size. The compiler version without pragmas needs over 150 lines of assembler code, while the version with pragmas needs around 50 lines. The compiler version with the software prefetch intrinsics is better than the standard versions. This could be caused by an earlier trigger of the prefetch stream. To support this suspicion a triad benchmark with different vector lengths was measured to compare two apart from software prefetch instructions identical assembler versions on the Pentium 4 test machine. The results are shown in Table 8.4. On this test machine the results confirm the suspect. The advantage for the software prefetch version increases for shorter vector lengths. This test was also repeated on other Intel tests machines (Core 2 and other Pentium 4 machines). Here the standard version showed better results for all vector lengths. A final confirmation can therefore not be drawn and further research is necessary to get more knowledge about the prefetch characteristics. For comparison two assembler versions are listed. There is still a large gap to the optimized assembly version, which uses the previously described block prefetch technique (see in Section 6.3.4). The basic algorithm for block prefetching reads as follows for this version of the triad:

Listing 8.6: Algorithm Block Prefetching

```c
for (k=0; k<SIZE; k+=BLOCKSIZE){
    prefetch_block(B,C);
    compute_block(A,B,C);
    A+=BLOCKSIZE;
    B+=BLOCKSIZE;
    C+=BLOCKSIZE;
}
```

The arithmetic code generated by the compiler is optimal. It can be difficult to convince the compiler to generate efficient prefetch code, and this is the only point where hand written code can improve the compiler generated code in this simple example.

<table>
<thead>
<tr>
<th>Test code</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>compiler w/o pragmas</td>
<td>4059 MByte/s</td>
</tr>
<tr>
<td>compiler w pragmas</td>
<td>4095 MByte/s</td>
</tr>
<tr>
<td>compiler w intrinsics</td>
<td>4129 MByte/s</td>
</tr>
<tr>
<td>assembly standard</td>
<td>4115 MByte/s</td>
</tr>
<tr>
<td>assembly optimized</td>
<td>4866 MByte/s</td>
</tr>
</tbody>
</table>
### Table 8.4: Software prefetch effect with different vector lengths

<table>
<thead>
<tr>
<th>Vector length</th>
<th>Standard version</th>
<th>Prefetched version</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>96 MB</td>
<td>4140 MByte/s</td>
<td>4158 MByte/s</td>
<td>18 MByte/s</td>
</tr>
<tr>
<td>192 MB</td>
<td>4140 MByte/s</td>
<td>4158 MByte/s</td>
<td>18 MByte/s</td>
</tr>
<tr>
<td>384 MB</td>
<td>4187 MByte/s</td>
<td>4187 MByte/s</td>
<td>17 MByte/s</td>
</tr>
<tr>
<td>768 MB</td>
<td>4202 MByte/s</td>
<td>4202 MByte/s</td>
<td>12 MByte/s</td>
</tr>
<tr>
<td>978 MB</td>
<td>4202 MByte/s</td>
<td>4202 MByte/s</td>
<td>12 MByte/s</td>
</tr>
</tbody>
</table>
Part III

Optimizing the multigrid algorithm on two selected architectures
Chapter 9

2D Red-Black Gauss-Seidel smoother on the x86/x86-64 Architecture

9.1 The x86/x86-64 architecture: An overview

The x86 architecture has its roots in Intel’s first 16 bit processor 8086. It has a CISC instruction set with variable instruction lengths. IBM 1981 presented the first Personal Computer using a modified version of this 8086 processor. It became the most widespread architecture and managed to defend its dominant position in the processor market till today. A main reason for its success is to always retain backward compatibility, and by that create an ever growing software base. Todays x86 processors are RISC/CISC hybrids, retaining its CISC instruction set but internally converting the CISC instructions into smaller RISC like micro instructions which are executed on a RISC CPU core. While this comes with the price of a higher hardware complexity the leading x86 processor manufacturers still managed to be competitive with regard to performance. Many see the x86 design as flawed, still the native RISC processors never managed to pull ahead significantly over a longer time period. Even newer approaches in the high end sector as the Intel IA64 are not able to keep an advantage with regard to performance, not to speak of price to performance or performance per watt ratios.

Major steps in the history of x86 are the introduction of 32 bit word length with 80386 in 1985 and a floating point unit on chip with the 80486 model in 1989. In 1996 Intel started to introduce the SIMD execution model with optional instruction set extensions to the core x86 architecture. As a natural step these extensions where included in the core architecture, when AMD introduced 64 bit word length with the next evolutionary step x86-64 in 2001. An overview about the x86 ISA can be found in Appendix D. Appendix E introduces the changes in the x86-64 architecture.
9.2 Test machine characteristics

Test machines in this work are an Intel Pentium 4 Prescott processor with 3.2 GHz (x86) and an AMD Athlon64 4000+ processor (x86-64) with 2.4 GHz. The operating system used on both machines is Linux using a 2.6 kernel. Appendix B gives an overview about the technical specifications of both platforms. While these two processors share many design features they have major differences affecting performance results. The focus of this work is how efficient our class of algorithms can be executed on modern cache based processors. The first step is to investigate what the processor is able to deliver at best. While the theoretical peak values are an upper bound already, even more significant are measured best case upper bounds. For our type of application the important values are peak double precision floating point calculation rate, peak memory bandwidth and peak cache bandwidth. To measure these values two benchmarks were implemented: The lssbench benchmark suite [Hau05], which measures cache characteristics and bandwidth and memory bandwidth for elementary memory operations, and the triad benchmark, which provides a number of assembly triad versions to measure memory and cache bandwidth. The triad benchmark also implements assembly snippets, which are supposed to reach peak performance on a processor.

9.2.1 Arithmetic Performance

Both architectures reach their peak floating point performance with a mixture of multiplication and addition operations. For this instruction mix both can execute two double precision operations per cycle. While the SSE instructions suggest to be executed in parallel, they have two operations per instruction, in reality the implementations at hand only have a 64 bit wide execution path in the SSE ALU. They can retire a SSE packed instruction only every second cycle. There is no special multiply add instruction in x86, to find independent multiplication and addition operations to be executed in one cycle is task of the hardware instruction scheduler.

The test code and detailed performance related issues can be found in Section 6.1. Modern out of order scheduled processors can reach around 90% of their peak arithmetic performance with one load stream from L1 cache. As can be seen in Table 6.2 the Athlon 64 can sustain this performance for two load streams, because its cache provides enough bandwidth. The Pentium 4 has with 6.4 GFlops/s a higher theoretical peak performance compared to the 4.8 GFlops/s of the Athlon 64 and saturates its L1 cache bandwidth already with 1 load stream. Both architecture show a cache bandwidth drop when including a store. This effect is larger on the Pentium 4.

9.2.2 Memory characteristics

Four cases are measured: memory load, memory store, memcopy and a full vector triad. Full means that three vectors are used for the triad. For the test machine architectures a technique called block prefetching has turned out to reach the best results. Details of this
measurements can be found in section 6.3. Important is to prefetch the data, which results in a significant advantage despite the fact that both processors have a hardware prefetch unit. Using the special prefetch hints introduced with SSE does not show the expected results for more complicated data access patterns. More effective on both processors is a technique called preloading, in which one load into a general purpose register per cacheline is executed. This together with the block prefetching, where loads and stores are separated is the best performing version for both test systems. Tests with regard to software prefetching instructions can also be found in section 8.2.

Table 9.1 gives an overview about the memory operation results. The Athlon 64 performs better overall due to its low latency on chip memory controller. One observation on the Pentium 4 is the low memory store performance. A satisfying explanation for the compared to the Athlon 64 low store bandwidth on all levels of the memory hierarchies was not found.

9.2.3 Cache characteristics

For cache blocking techniques it is important to know how much the cache bandwidth decreases for different levels of the cache hierarchy. Measured are cache load (Graph 9.1 a) and cache store (Graph 9.1 b) bandwidth. Moreover an in cache version of the triad benchmark is tested (Graph 9.2). The Athlon64 has a larger L1 cache size compared to the Pentium 4 with L1 cache size 64 kB to 16 kB on the Pentium 4. The L1 bandwidth is for both CPUs on a comparable level. The Athlon 64 reaches its highest L1 cache bandwidth with the register/memory arithmetics instruction format, while the register/register arithmetic instructions with separate load instructions reach significantly lower cache bandwidth (more details on this issue is in Section 7.2). The Pentium 4 has a higher L2 bandwidth. L1 and L2 are connected by an 256 bit wide bus, which can deliver up to 100 GB on our test machine. On the Pentium 4 the L2 Data cache is the primary target cache. Store performance is drastically lower due to the write-through cache architecture. This still does not explain the also lower L2 store bandwidth. The Athlon 64 has a low bandwidth but also low latency L2 Data Cache. The L2 cache on the Athlon 64 is non-inclusive. It reaches lower bandwidths but as a write back-cache reaches the same load and store performance in L1 and L2 cache. The triad (Graph 9.2) for the optimized version using load-execute instructions (Athlon64 opt) showed the same bandwidth as with 8 byte loads. The Pentium 4 can sustain only half of the the bandwidth

<table>
<thead>
<tr>
<th></th>
<th>AMD Athlon64 4000+</th>
<th>Intel Pentium 4 3.2 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>memread</td>
<td>6096 MByte/s</td>
<td>5810 MByte/s</td>
</tr>
<tr>
<td>memwrite</td>
<td>6058 MByte/s</td>
<td>4018 MByte/s</td>
</tr>
<tr>
<td>memcpy</td>
<td>6035 MByte/s</td>
<td>5651 MByte/s</td>
</tr>
<tr>
<td>triad</td>
<td>5163 MByte/s</td>
<td>4880 MByte/s</td>
</tr>
</tbody>
</table>

Table 9.1: Measured memory bandwidth
Figure 9.1: Measured cache bandwidth
on all cache levels when using 8 byte loads compared to 16 byte loads. In the triad results the Pentium 4 benefits from its strong connected L2 cache reaching a sustained bandwidth of over 20 MBytes/s. The Athlon 64 shows a high bandwidth in the L1 cache but has a large drop to the weakly connected L2 cache. Overall it can be said that the Athlon 64 is a low latency memory architecture, leading to good memory performance with legacy code and reaching a good memory bandwidth with unoptimized and unstructured code. The Pentium 4 is a high bandwidth but also high latency architecture tuned on streaming applications, but relies on SSE code together with proper prefetching to reach its potential. A major advantage in the context of cache blocking is its strongly connected L2 Data cache.

9.3 Optimization techniques

**Data Layout** One point previous work clearly showed was, that it is crucial to choose a suitable data layout for a given architecture. For cache based processors this means to maximize data access temporal and spatial locality. Modern architectures share many aspects which are also important for vector processors. It must be possible to stream the data in a consecutive data access order. For a red black Gauss-Seidel smoother used in geometric multigrid this means, that for using the packed arithmetic instructions together with 16 byte loads red and black points have to be stored into separate arrays. This was already suggested in a paper on optimizing multigrid on a classical vector machine [Bra83]. Also important in this context is the alignment of the data arrays. To use 16 byte moves the data has to be 16
byte aligned, this is also called natural alignment which means that the data is aligned at a memory address which is a multiple of its own size.

The requirements to the data layout are:

1. pairs of a vector should be stored sequentially in memory

2. as many vectors as possible should be naturally aligned, in our case 16 byte

Figure 9.3 shows the splitting of the unknown vector into two arrays, one for the black and one for the red points. In Figure 9.4 the padding necessary to enable 16 byte aligned data accesses are shown. The consequences of the data layout on the five point stencils resulting from the Red-Black Gauss-Seidel algorithm with regard to aligned and unaligned data accesses are shown in Figure 9.5. It can be seen that the majority of accesses are naturally aligned.

**Code level optimizations** The smoother code was implemented in assembly to prevent an influence of the compiler and enable a direct view on the optimizations. Besides the decision what instructions to use one crucial task is register scheduling. On x86 with the limited number of 8 general purpose registers it is difficult to implement the nested loop structures together with the addressing of multiple multi dimensional arrays without register spilling and too much address calculations in the innermost loop body. There are two basic strategies for optimizing this algorithm on our test machines which are contradicting. First it can be implemented as a streaming algorithm (vector like), trying to reach a high sustained memory bandwidth to increase performance. Second cache blocking can be applied to increase temporal and spatial locality in order to lower the pressure on the memory bus and enable a more efficient usage of the caches. For both strategies it is necessary to implement all floating point calculations using the packed SSE instructions in order to reach good performance.

For the streaming version it is important to choose the right instructions for load and stores to memory. For loads it is important to use the 16 byte load instructions loading data into one XMM register with one instruction. Even more important is to use the non temporal store

![Figure 9.3: Separation of black and red values](image-url)
Figure 9.4: Layout of red and black arrays x=9

Figure 9.5: Discrete five point stencil
instructions (also called streaming stores), which can be write combined by the implementation and enable stores to main memory in larger data junks. These instructions are only hints, the actual implementation is not guaranteed. On our two test machines these special stores showed by far the best store performance.

**Cache blocking** For a basic introduction to blocking techniques we refer to [Kow04] and [Wei01]. For the Red-Black Gauss-Seidel smoother a standard rectangular blocking technique was chosen. Important is to choose a technique which still obeys the assumptions of the hardware prefetch unit. Difficulties arise also due to the separated arrays. The basic procedure is illustrated in Figures 9.6 and 9.7. The 2D grid is cut into stripes. Each stripe is updated from top to bottom and from left to right line wise. Inside a sub block the values are updated line wise. A stripe is supposed to fit in L2 cache while the rectangles being currently updated have to fit into L1 cache. This scheme can therefore be seen as a two stage blocking scheme for L2 and L1 caches. Using this technique the data is still fetched from memory in a linear way, without cutting of consecutive memory arrays and jumping in address space in order to support the work of the hardware prefetch unit.

**Software prefetching** The effectiveness of the hardware prefetch unit can be increased by using prefetch instructions. On the covered architectures data prefetching implemented by executing one load instruction per cacheline has shown to increase the sustained main memory bandwidth.

### 9.4 Results

In the preceding work of Christian Weiss [Wei01] a modified version of the standard constant coefficient Red-Black Gauss-Seidel Update was chosen:

\[
U_{x,y} = c_N \cdot U_{x,y-1} + c_S \cdot U_{x,y+1} + c_C \cdot U_{x+1,y} + c_E \cdot U_{x+1,y} + c_W \cdot U_{x-1,y} + F_{x,y} \tag{9.1}
\]

This is a modified variable coefficient update, because all coefficient are constant values. The results presented in this chapter use the above modified update scheme, without the central point coefficient though.

**Dime 2D Results** This work is a continuation of previous dissertations. As identified in section 8.1 the results at that time were influenced by inefficient compiler generated code. Recent Intel compilers manage to produce efficient instruction code for the Dime Fortran 77 codes. In Figure 9.8 the original dime 2D Codes compiled with a recent compiler (Intel icc 9.0) are presented for the two test machines. The best optimized version reaches a speedup up to factor 3 against a standard implementation. The plain version refers to the rb1 version in the dissertation of Christian Weiss [Wei01] with one sweep per half update. The result plain is the rb2 version in Christian Weiss’s notation with a fused update. The blocked versions
Figure 9.6: Global update scheme for blocking technique

Figure 9.7: Fusion of two full iterations with a 8x2 block in 2D
refer to the rb7 version with two, three and four way time blocking. These results together with the standard C implementation are the baseline for further benchmarks. The complete Dime 2D results for all code versions and a detailed explanation of the code versions are listed in appendix C.

**Optimized Data Layout**  Figure 9.9 show a standard C implementation against an assembler implementation with optimized data layout. In the assembler version special care was taken for good register scheduling and vectorization. It has to be emphasized that is not the purpose of this work to compare compiler generated code against handwritten assembly code. The question whether a compiler could also apply these optimizations is out of scope of this work. Also the question if the techniques described here could be implemented in a high level language with the same performance is not part of this work. Still as a frame of reference the standard C implementation is listed for comparison. The C code was compiled with version 9.0 of the Intel C compiler. For the assembly version one with and one without software prefetching is listed. In figure 9.9 a fully vectorized version with 16 byte loads and with temporal stores was used. The baseline C performance of the Athlon 64 is better than of the Pentium 4 reaching 831 against 675 MFlops/s. Also software prefetching has a much higher influence on the Athlon 64 with 1312 against 1040 MFlops/s. For the in cache performance the low L2 cache bandwidth can be seen for the Athlon 64 compared to the Pentium 4 with L2 performance around 2600 against 3400 MFlops/s.

**Streaming Implementation**  In figure 9.10 the full streaming implementation for both architectures is shown. In these versions non temporal store instructions are used. Both machines show a large increase in performance. Especially the Athlon 64 shows a very good performance due to its good memory bandwidth. The Pentium 4 suffers from its low memory store bandwidth. The in cache performance is reduced drastically by using the non temporal stores, which bypasses the caches. Especially for this setting the Athlon 64 benefits significantly from software prefetching. In memory the Athlon 64 reaches 1780 against 1379 MFlops on the Pentium 4, both machines reached their best results with software prefetching. The streaming approach results in an improvement by a factor of 2 against the standard c implementation on both machines. The effect of the performance drops on the Athlon 64 were not further investigated. The streaming implementation on the Athlon 64 already reaches higher results than the blocked Dime version in figure 9.8.

**Blocked Implementation**  Finally in figure 9.11 the results of the described blocking technique can be seen. For this technique the Pentium 4 is the better suited architecture with its strongly connected L2 cache. For the Athlon 64 no big improvement can be reached with
Figure 9.8: Results for optimized Dime Fortran77 codes.
CHAPTER 9

blocking two iterations against blocking three iterations in one sweep. This suggests that the Athlon 64 already suffers from L2 bandwidth limitations. The Athlon 64 reaches 2557 MFlops/s from main memory while the Pentium 4 can maintain 2961 MFlops/s on a stable level. This results in improvements by factor of 3 on the Athlon 64 and a factor of above 4 for the Pentium 4 against the reference C implementation.

9.5 Validation of the Results

The Athlon 64 reaches better results with unoptimized code than the Pentium 4. Target cache is the L1 cache. The Pentium 4 on the other hand uses its small L1 cache as a transit cache with the L2 cache being the primary target cache. The Pentium 4 architecture reaches a high performance with optimized code. The strong cache subsystem make it well suited for cache blocking techniques. This is valid as long the packed SSE instructions are used and a global data access pattern which is suited for the hardware prefetcher is maintained. The Athlon 64 in turn reaches good results with a streaming implementation. This is caused by its good main memory performance also seen in the micro benchmarks. The Pentium 4 with the streaming implementation suffers from its low main memory store performance. The optimizations enabled an efficiency of over 40% of peak on the Pentium 4 and over 50% of peak on the Athlon 64. These results show that modern architectures can reach a reasonable efficiency for our class of algorithms taken into account that this is an initially main memory bandwidth limited algorithm. A similar work on the example of the Lattice Boltzmann algorithm can be found in [Hau05, THR05].
Figure 9.9: Standard assembler implementation with optimized data layout
Figure 9.10: Streaming assembler implementation
Figure 9.11: Blocked assembler implementation
Chapter 10

3D Geometric Multigrid on the IA64 Architecture

The optimized 3D versions do not perform as well as the 2D versions on the x86/x86-64 architectures versions \cite{Stue05}. The algorithmic balance characteristics are similar for the 2D and 3D case, still for the 3D case the overall code complexity is higher. There are more memory streams, memory addressing is more complex and there are more calculations to be executed in the innermost computational kernel. The suspicion is, that the 3D version suffers from hardware bottlenecks present on common x86/x86-64 architectures. With IA64 an alternative architecture which has less hardware bottlenecks and provides enough resources to run also the 3D code well is available. Its software oriented architecture provides many opportunities for low level optimization. Therefore it was chosen as platform to implement a highly optimized 3D multigrid code. In difference to the optimized implementation on the x86/x86-64 architecture (see Chapter 9) not only the smoother routine is optimized. Instead an integrated optimization approach for the complete multigrid algorithm was implemented. For complete results refer to \cite{Stue06, STR08, STR06}.

10.1 The IA64 Architecture: A Short Overview

The IA64 architecture is a 64 bit architecture cooperatively developed by Intel and HP. It was intended to found a new era in computer architecture replacing the established RISC processors. While in RISC processors ILP is exploited dynamically at runtime by the hardware IA64 uses a explicit software approach to exploit available ILP. The IA64 architecture is also often referred to as Explicit Parallel Instruction Computing (EPIC). EPIC tried to overcome the rising hardware complexity of out of order processors by explicitly encoding multiple operations per instruction. It is a trade off to decrease hardware complexity by increasing compiler complexity. While the ISA was designed to suite a compiler the high software complexity gives also a lot of opportunities to an assembler programmer. The processor is very predictable on machine code level, allowing an analytical approach during implementation. This is in con-
trast to the black box character of out of order scheduled processors. The available transistors on the Itanium 2, the only hardware implementation of the IA64 architecture, are used for elementary hardware resources as caches and register banks. The IA64 architecture has fewer hardware bottlenecks than other processors and is therefore well suited for a highly efficient implementations of the 3D multigrid algorithm, which never performed as well on the out of order processors as for the 2D case. The only available implementation of this the Intel Itanium processor, is in order execution. The IA64 architecture has a complex with many novel concepts not found on common architectures. In the following a brief introduction to those features is presented, which are relevant for our class of algorithms. For further reading refer to the Intel manuals \[Int06a\], \[Int06c\] and \[Int06b\].

10.1.1 Instruction Formats

A very long instruction word consists of up to three instructions, this is also called a bundle. The instruction stream is divided in instruction groups by stops (indicated with \(;;\) in the assembler code). Inside an instruction group the processor can execute all instructions in parallel. If a specific mix of instructions can be mapped on the processor hardware is decided by the hardware at runtime. To allow parallel execution Read After Write (RAW) and Write after Write (WAW) dependencies inside an instruction group are not allowed, while Write after Read (WAR) is allowed. There exists a limited number of templates for an instruction bundle. These templates specify where stops are allowed and of which combinations of instructions a bundle can consist. Instruction categories are listed in Table 10.1.

The following slot combinations of instructions are allowed: MII, MMI, MMF, MIB, MBB, BBB, MMB, MFB and MLX. A stop can occur at the end of a bundle or between the I slots in the MII bundle or M slots in the MMI bundle. If the compiler is not able to fill the bundles with sensible instructions it has to insert nop instructions, which stands for no operation. These nop instructions are executed in current hardware implementations, this means they really block an execution unit.

<table>
<thead>
<tr>
<th>Instruction Category</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory instructions</td>
<td>M</td>
</tr>
<tr>
<td>floating point instructions</td>
<td>F</td>
</tr>
<tr>
<td>complex integer instructions</td>
<td>I</td>
</tr>
<tr>
<td>branch instructions</td>
<td>B</td>
</tr>
<tr>
<td>simple integer instructions</td>
<td>A</td>
</tr>
<tr>
<td>special instructions</td>
<td>LX</td>
</tr>
</tbody>
</table>

Table 10.1: Instruction Categories on the IA64 Architecture
10.1.2 Register Set

The IA64 architecture has 128 64bit wide general purpose registers. Registers r32 to r127 are organized as a register stack avoiding the spilling and filling among procedure calls. There are 128 82bit wide floating point registers, from which registers fr32 to fr127 are organized as rotating registers. Additionally the IA64 architecture has a special register file with 64 1bit predicate registers for the execution of conditional code, from which registers pr16 to pr63 are organized as rotating registers. There are several other register files.

10.1.3 Selected Special Concepts

IA64 tries to address several problems that are apparent on modern out of order scheduled suberscalar processors. The new techniques it introduces on instruction set level are targeted to improve instruction level parallelism and hide existing latencies. Among those new techniques are:

- Explicit data and control speculation to expose available ILP by hiding latencies of operations
- Predication to remove the impact of branches
- Software pipelined loops to reduce loop overhead
- Software prefetching

Some of the more relevant features for scientific computing are introduced in the following. The IA64 assembler is complex and it is not possible to introduce it in the scope of this work in depth. Still the examples should show the concepts without the need to understand every detail.

**Predication** Traditionally decisions are handles by conditional branching. Typically based on the result of an instruction bits are set in the status word. Jump instructions are executed depending on the bit values in the status word. An if-else code block is written in C:

```
if (a == 5)
   a=0;
else
   a++;  
```

In x86 assembler code this can be written as:

```
cmp eax, 5
jne .L2
mov eax, 0
```

Listing 10.1: If else branch

Listing 10.2: If else branch in x86 assembler (Conditional Branching)
The compare instruction `cmp` sets bits which the `jne` instruction evaluates. If the condition is true the jump is not executed. After the branch is executed it has to branch to the end label `.L6`. If the comparison evaluates to true the jump to label `.L2` is executed. The else branch is executed.

On an architecture supporting predication instructions can be predicated, which means their execution depends on a predicate set before. On the processor all slots are executed but depending on the predicate bit they do not alter the state. Thereby control dependencies are converted into data dependencies. Advantage of such an approach is, that branches over small code sections interfering with pipelining can be avoided. The code is in principle more compact. The predicated instructions can be scheduled with other independent instruction and thereby increase the potential of ILP. An if/else block could be implemented as follows on IA64:

Listings 10.3: If else branch in simplified IA64 assembler (Conditional Branching)

```assembly
1 cmp.eq p1, p2 = 5, r32 // p1 is set according to the comparison, p2 to NOT p1
2 ;; // a stop is necessary, before reading p1,p2
3 (p1) add r32 = 0, r0 // small constants are fastest created by addition
4 (p2) add r32 = 1, r32 // with constant register r0
```

Software Pipelined Loops The execution of loops involves a large overhead during in order execution. A classical for loop multiplying a double vector with a scalar is written in C as:

Listing 10.4: For branch in C

```c
1 for (i=0; i<n; i++){
2   a[i] = a[i] * c;
3 }
```

An implementation in x86 assembler using SSE instructions:

Listing 10.5: For loop in x86 assembler

```assembly
1 jmp .L2
2 .L3:
3 movapd xmm0, [a+eax*8]
4 mulpd xmm0, xmm1
5 movapd [a+eax*8], xmm0
6 add eax, 2
7 .L2:
8 cmp eax, ecx
```
Jump to label .L2 to check the condition on loop entry. If counter register eax is less than stopping criterion in register ecx the jump to .L3 is executed. The loop body consists of loading data into xmm0 addressed with the counter register, multiplying it with the constant loaded before into xmm1 and storing it back to memory. The counter register is incremented and the jump condition checked again.

The three instructions add, cmp and jl are with regard to to the arithmetic operations overhead. For three useful instructions three overhead instructions are executed every iteration. Moreover the multiply depends on the data be loaded into the xmm0 register. With no optimization this will need at least 2-3 cycles. The store relies on the result of the multiply to be available. In this time the pipeline is stalled. The first optimization is to unroll the loop to reduce the loop instruction overhead. With an unrolling of the loop by a factor of four in every iteration 12 useful instructions are executed against 3 overhead instructions. In principle this operation could from the data dependency point of view be executed in one step. Modern out of order processors with speculative execution try to exploit this available parallelism. They exploit available ILP by speculatively executing the loads in advance and delaying the arithmetic instructions. By that they can overlap the execution and keep the pipeline busy. Unrolling is already a way to implement software pipelining. For the following example it is assumed that each floating point instruction takes 3 cycles, the loop overhead is neglected for the moment. The unoptimized loop iteration will take 3 cycles for the load, 3 cycles for the execution and 1 cycle for the store. Because the next iteration can already execute the load, it is not depending on the store. This means altogether at least 7 cycles for one iteration. A three time unrolled version can be seen in the following:

```assembly
1  jmp  .L2
2  .L3:
3  movapd xmm0, [a+eax*8]
4  movapd xmm1, [a+eax*8+8]
5  movapd xmm2, [a+eax*8+16]
6  mulpd xmm0, xmm1
7  mulpd xmm1, xmm1
8  mulpd xmm2, xmm1
9  movapd [a+eax*8], xmm0
10  movapd [a+eax*8+8], xmm1
11  movapd [a+eax*8+16], xmm2
12  add  eax, 6
13  .L2:
14  cmp  eax, ecx
15  jl   .L3
```

Listing 10.6: Unrolled for loop in x86 assembler
With this implementation the dependencies are hidden. An instruction can be executed in every cycle. This means that for 3 iterations 9 cycles are needed, 3 cycles per iteration. This works well as long as each instruction takes the same time to finish. Consider that the load takes a longer time to finish, for example 10 cycles. Now the loop has to be unrolled 10 times to hide this dependency. This would involve a prolog before the actual loop to check, if the number of iterations is dividable by 10 and if not to introduce special treatment instructions for this case. In short a lot of overhead for the implementation of the software pipelined loop is necessary probably compensating the advantage. Consider another implementation. In the following the code is abstracted. L denotes the load, M the multiply and S the store.

Listing 10.7: Software pipelined for loop

```plaintext
1 Prolog
2
3 for (i= 1 to (END - 4))
4   L(i+4)
5   M(i+1)
6   S(i)
7
8 Epilog
```

In an iteration view this reads as:

- **Iteration 1:** L(5) M(2) S(1)
- **Iteration 2:** L(6) M(3) S(2)
- **Iteration 3:** L(7) M(4) S(3)
- **Iteration 4:** L(8) M(5) S(4)
- **Iteration 5:** L(9) M(6) S(5)
- **Iteration 6:** L(10) M(7) S(6)
- **Iteration 7:** L(11) M(8) S(7)

Between the load in iteration 1 of the loop body and the multiplication of the loaded data are 10 instructions. The instructions in between are filled with calculations for previous iterations. The loop body is in full pipelined execution, the pipeline is filled in the Prolog and drained in the Epilog. For this example the Prolog and Epilog are:

Listing 10.8: Software pipelined for loop: Prolog

```plaintext
1 L(1)
2 L(2)
3 L(3)
4 L(4) M(1)
```

Listing 10.9: Software pipelined for loop: Epilog

```plaintext
1 M(END -2) S(END -3)
2 M(END -1) S(END -2)
3 M(END) S(END -1)
```
Epilog and Prolog add a significant amount of overhead. Besides this overhead a major problem with software pipelined loops is register scheduling. Because the multiply expects the values from a previous iteration the values loaded there have to be moved to a different register to not be overwritten by the load in this iteration. Of course these register moves add additional dependencies and scheduling gets more complicated. If register to register moves are an expensive operation it may be very difficult to find a good solution. Software pipelined loops are in theory an effective optimization, the actual implementation poses many problems on real architectures.

The IA64 architecture offers hardware support for software pipelined loops making their implementation easier. It addresses the problems described above by:

- Rotating register banks: Registers are redirected to different register by hardware in each iteration of a loop. This makes additional move instructions obsolete.

- Predicates which take their values from special loop instructions make a separate Prolog and Epilog unnecessary.

- Special loop instruction support

The IA64 architecture has special application registers for loops:

- Loop counter \( \text{ar}\.\text{lc} \): Is decremented by the branch instructions \( \text{br}\.\text{ctop} \). \( \text{br}\.\text{ctop} \) branches if \( \text{ar}\.\text{lc} \) is larger than zero.

- Epilog counter \( \text{ar}\.\text{ec} \): Controls the correct predicate register setting during Epilog.

The latencies are different on the Itanium 2, for consistency it is assumed that the latencies are the same as in the example above. An implementation of above example in IA64 assembler is as follows:

Listing 10.10: Software pipelined for loop: IA64

```assembly
1 add r2 = -1, r33
2 mov pr.rot = 1 << 16 //this sets p16 to true and p17 to p63 to false
3 mov ar.ec = 5 ;; //number of pipeline stages
4 add r33 = 0, r32 //delayed store
5 mov ar.lc = r2 ;; //initialize loop counter
6 .pipelinedloop:
7 (p16) ldfd f32 = [ r32 ],8
8 (p19) fmpy f35 = f35, f8
9 (p20) stfd [ r33 ] = f36,8
10 br.ctop.sptk .pipelinedloop
```

The main work is accomplished in the \( \text{br}\.\text{ctop} \) instruction. The algorithm implemented by this instruction is illustrated in figure 10.1. In the following the code in listing 10.10 is explained in detail. In the code section before the loop the loop counter \( \text{ar}\.\text{lc} \) and the rotating
Predicate registers are initialized. The register \( ar.ec \) is set to the number of stages. After loop entry three stages are run through:

1. **Prolog**

   On loop entry only predicate \( p16 \) is set to true. Therefore in the first iteration only \( p16 \) is executed. The branch instruction checks if \( ar.lc > 0 \). This is true, therefore the loop counter is decremented. The last rotating predicate register \( p63 \) is set to true. While register rotation \( p63 \) will get \( p16 \). At last the actual jump is executed. This continues until the last predicate \( p20 \) is set to true. From this moment the pipeline is full and all predicated instructions get executed in each iteration.

2. **Loop**

   The relevant predicates are all true and the loop counter get decremented until it is zero. The floating point registers also rotate. That means, that \( f32 \) will be \( f35 \) three iterations later.

3. **Epilog**

   The Epilog is entered as soon as the loop counter is zero. As long as the Epilog counter \( ar.ec \) is greater than zero the Epilog counter is decremented and the predicate \( p63 \) is set to false. Again during register rotation \( p63 \) gets \( p16 \). At the end the jump is executed. This means that in the first iteration of the Epilog the load instruction is not executed. This continues until the Epilog counter is zero. Now all relevant predicate registers are false. The loop is left.

A comparison of the Prolog and Epilog procedure with the handwritten ones in \( 10.8 \) and \( 10.9 \) show that exactly the same instructions are executed. A good scheduling and resource usage
Software Prefetching  To reduce memory access latency and memory bandwidth issues the IA64 architecture supports many instructions controlling the behavior of the memory subsystem. A central role has the \texttt{lfetch} prefetch instruction to prefetch a cacheline from memory into cache. This instruction has special annotations controlling in which cache level the data shall be fetched. Further loads and stores can be marked as non temporal, this data gets stored in special parts of the cache which are replaced faster. As on the x86 architecture these instructions are hint instructions. On the Itanium 2 though they work reliably and effectively. Especially in software pipelined loops prefetch instructions can be exactly scheduled and wait periods can be inserted in the pipeline without a performance penalty. A common technique in combination with software pipelined loops is to fetch several data streams with one prefetch instruction.

\begin{verbatim}
Listing 10.11: Prefetching multiple streams in software pipelined loops
1 add r34 = 900, r15  // Prefetch 900 bytes in advance
2 add r33 = 908, r14
3 .loop:
4 ...
5 lfetch.fault.nt1 [ r34 ] // Load the cacheline containing the data
6          // at address r34 into level 2 cache
7 ...
8 add r32 = 16, r34
9 ...
10 br.ctop.sptk .loop
\end{verbatim}

This technique makes use of the rotating general purpose registers.

10.2 Optimization Techniques

10.2.1 Loop Optimizations

A critical point for efficient loops aside from software pipelining in IA64 is a good mapping of instructions on bundles. The loop has to be unrolled to find an ideal amount of instructions, which can be mapped in a way that all bundles are fully utilized without wasting unnecessary cycle.

Software Pipelined Loops  Important points with regard to constructing software pipelined loops are:

- Loop body shall take more than one cycle to enable better branch prediction
- Enable parallel loads to save load instructions while maintaining maximum bandwidth
- Enable better bundling and manage to execute the optimum two bundles in one cycle
• Reduce the number of necessary prefetch instructions

Register Reuse  Often values are used in successive calculations. Here the rotating registers in software pipelined loops can be used to enable a cheap reuse without the need of additional instructions. Especially with stencil based computations this opportunity is often available, as for example in one dimensional linear interpolation. One dimensional linear interpolation is written as $B[n] = (A[n - 1] + A[n + 1])/2.0$, in which the value $A[n]$ contributes to $B[n - 1]$ and to $B[n + 1]$. Code example 10.12 illustrates this technique. The vector $A$ is only loaded one time. Because their values are rotated through the rotating register set they can be used multiple times without the need for additional loads or register to register moves. Vector $A$ is loaded in register $f32$. In the add instruction the values in the registers $f35$ and $f37$ are values three and five iterations ago loaded into $f32$.

Listing 10.12: Register reuse in software pipelined loops

```
1 // r32 Zeiger auf A[n-1]
2 // r33 Zeiger auf B[n]
3 // f8 = 0.5
4 ld fd f34 = [ r32 ], 8
5 ld fd f33 = [ r32 ], 8
6 ;;
7 .loop :
8 (p16) ld fd f32 = [ r32 ], 8
10 (p21) fmpy f52 = f52, f8
11 (p23) stfd [ r33 ] = f54, 8
12 br.ctop.sptk .loop
```

Overlapping of Software Pipelined Loops  Software pipelined loops add an overhead to the loop by the need for a Prolog and Epilog phase. This overhead can make their application unattractive for short loop lengths. Three cases can be distinguished:

1. High number of iterations: The overhead of Epilog and Prolog can be neglected.

2. Medium number of iterations: Overhead is significant. Still complete unrolling is not yet attractive.

3. Small number of iterations: The loop can be completely unrolled.

For the second case the execution of the Prolog and Epilog can be overlapped if a loop is nested and executed multiple times as apparent in many scientific codes. This is achieved by setting the Epilog counter to 1 at the end of the loop core and thereby preventing the execution of the Epilog. When the same loop body gets executed again, after reinitializing the first pipeline stages, the loop counter and setting the Epilog counter to 1 again, Prolog of the current loop sweep and the Epilog of the previous loop sweep are executed at the same time. This technique can be used as long as the following conditions are ensured:
• All operands beginning at the second pipeline stage are invariant, used exclusive by every stage or handed through in rotating registers.

• Operands are not reused or used delayed.

• It must be assured that the Prolog does not load data from memory, which may not yet be stored by the Epilog.

There are possibilities, that allow to combine loop overlapping and register reuse explained in [Stü06].

**Anticipatory Prefetching**  With the prefetching technique explained in Section 10.1.3 the data is loaded in advance with a constant distance. That means that the prefetch area and the operator arithmetic area are shifted. With long running loops these two areas almost completely overlap. For a medium number of iterations the overlap is in relation smaller. Data is prefetched which is not used and bandwidth is wasted. For this case a technique can be used that anticipates jumps in the operand area and adapts the prefetch stream also to subsequent loop sweeps. Figure 10.2 illustrates the prefetch pattern with long running loops without jumps. The next line shows long running loops with small jumps in the stream. Here the data which is prefetched to far is then used by later accesses. With small loop lengths together with large jumps in the worst case the streams do not overlap at all. In the time it takes until the prefetch stream could reach the operand data stream the data stream already jumps. With anticipatory prefetching the prefetch pattern is adapted to known jumps and the overlap is maximized even for short loops.

**10.2.2 Optimizations on Algorithmic Level**

Starting point is a characterization of the geometric multigrid algorithm (in our case with a $V(2,2)$-cycle) with regard to arithmetic operations and memory accesses. It is assumed, that a coarsening of the grid has $\frac{1}{8}$ of the $n^3$ unknowns. The memory requirements of the boundary
nodes are not taken into account. Table 10.2 shows the operations and memory accesses for a grid in relation to the number of unknowns. In this table also the load operations needed by write misses are taken into account. Listed are the common optimizations to combine the calculation of the residual and the restriction and an immediate correction with the interpolated fields. The geometric sum \( \sum_{k=0}^{\infty} \left( \frac{1}{8} \right)^k = \frac{8}{7} \) can in good approximation be taken to get a good estimation on the distribution of cost on different grid levels. 87.5 % of arithmetic operations and memory traffic are spent on the finest grid level. Table 10.3 shows the distribution on the components of a V(2,2)-cycle.

Already with only two pre- and post-smoothing steps the smoother dominates. But also the calculation of the norms is remarkable costly.

**Algorithmic Optimization** The Red-Black Gauss Seidel method, used as smoother, sets the values in each iteration first on all red and later on all black points so that the residual is zero. If at least one pre-smoothing iteration is performed the calculation of the residual can be omitted because it is already known that they are zero. The calculation of the residual and restriction can therefore be restricted to the red points of the coarse grid. Each new result moreover only depends on the surrounding points and the right hand side of the equation, but not on previous results. If at least one post-smoothing step is performed, therefore interpolation and correction can be restricted to the black points of the fine grid.

**Data layout** Here the same data layout as proposed in Section 9.3 is used. The concept is extended to 3D as explained in [Stu05] and [Stu06]. This results in two half grids. Each half grid is aligned to a 16 byte boundary and all lines are padded to an equal and even length. Advantage of this data layout are:

- Separate access on red and black points.
- More efficient cache usage.
- Allows IA64 parallel loads saving instructions and enabling higher L2 cache bandwidth.

A disadvantage is the more complex addressing and an increase of data streams.

**Optimizations of loop kernel** If values are calculated for 2 \( \times \) 2 lines in one sweep this leads to a reduction of memory and arithmetic operations. By using parallel loads and the reuse of values in rotating registers the calculation of eight unknowns can be performed with only 4 single and 14 parallel loads. The number of store operations stays at 8.

By reusing intermediate results arithmetic calculations can be saved. For the calculation of eight unknowns this reduces the number of arithmetic operations from 56 to 52. With above techniques there are enough free slots in the instruction bundles to implement anticipating prefetching and overlapped loop execution.

Above method can be seen as implicit spatial blocking, in which two planes are calculated at once.
Table 10.2: Estimate of flops and memory accesses per unknown on one grid level

<table>
<thead>
<tr>
<th>Component</th>
<th>Flops</th>
<th>Memory throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Simple</td>
<td>Itanium</td>
</tr>
<tr>
<td>Smoother $^1$</td>
<td>$(i+j) \cdot 8$</td>
<td>$(i+j) \cdot 7$</td>
</tr>
<tr>
<td>Residual</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Restriction</td>
<td>$\frac{39}{8}$</td>
<td>$\frac{27}{8}$</td>
</tr>
<tr>
<td>Interpolation</td>
<td>$\frac{52}{8}$</td>
<td>$\frac{52}{8}$</td>
</tr>
<tr>
<td>Correction</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Residual Norm $^4$</td>
<td>11+?</td>
<td>10</td>
</tr>
<tr>
<td>Residual &amp; Restriction</td>
<td>$9 + \frac{39}{8}$</td>
<td>$8 + \frac{27}{8}$</td>
</tr>
<tr>
<td>Interpolation &amp; Correction</td>
<td>$\frac{52}{8} + 1$</td>
<td>$\frac{52}{8}$</td>
</tr>
</tbody>
</table>

$^1$ i Pre-smoothing- and j Post-smoothing-Iterations

$^2$ between 4 and $4 \cdot (i+j)$ for $i > 0$ and $j > 0$

$^3$ between 2 and $2 \cdot (i+j)$ for $i > 0$ and $j > 0$

$^4$ Calculation only sensible on finest grid and only necessary for one norm

Table 10.3: Distribution on the components for a V(2,2)-cycle

<table>
<thead>
<tr>
<th>Component</th>
<th>Flops (Itanium)</th>
<th>Memory-Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smoother</td>
<td>51 %</td>
<td>66 %</td>
</tr>
<tr>
<td>Residual &amp; Restriction</td>
<td>21 %</td>
<td>12 %</td>
</tr>
<tr>
<td>Interpolation &amp; Correction</td>
<td>12 %</td>
<td>12 %</td>
</tr>
<tr>
<td>Residual Norm</td>
<td>16 %</td>
<td>10 %</td>
</tr>
<tr>
<td>Number per unknown</td>
<td>ca. 62,5</td>
<td>ca. 20,7</td>
</tr>
</tbody>
</table>

99
CHAPTER 10

**Blocking** A major potential in the context of iterative algorithms is a reduction of the memory transfers by temporal blocking over multiple iterations. For the Itanium large blocks of simple structure were chosen, a pair of two planes, as primary block size. With blocking $i$ iterations (one iteration consists already of $2$ half iterations) first in $2 \times i$ planes the first red half iteration and in $2 \times i - 1$ half iterations also the first black half iteration is performed. If $i > 1$ this process is repeated until in plane $1$ the $i$-th black half iteration is finished \((10.3 a)\). From there on the red half iteration in plane $2 \times i + 1$ and $2 \times i + 2$, then the black half iteration in planes $2 \times i$ and $2 \times i + 1$ and so on can be calculated. The block with the first red half iterations is shifted by two planes. Between the finished planes and the planes which are not updated a stair of subsequent updates is formed \((10.3 b)\). At the end, according to the setup of the stair structure the calculation must be finished \((10.3 z)\).

If the size of the planes and the number of blocked iterations is small enough, all needed data can be completely kept in cache. Depending on the number of iterations, the cache and plane size it might be necessary to introduce a two stage blocking method. This second blocking plane cuts the planes from the standard method into several sub blocks. Inside these sub blocks the standard blocking method described above can be applied. Of course at the cut boundaries also a special boundary treatment has to be performed \((10.4)\).

**Enhanced Prefetching** Temporal blocked code has fluctuating memory bandwidth requirements. New values have to be fetched from memory as the update frontier is moving. This effect is increased by cacheline replacement due to conflict misses. In this implementation a special anticipating prefetch strategy that not only prefetches data needed in near future in the L2 cache, but also fetches data needed in future iterations in the lowest cache level. These prefetch operations are issued in a way, that memory accesses are distributed equally over iterations.

**Parallelization** Despite the fact, that this thesis limits itself to single processor optimizations, also thread level parallelization results are presented here. The used machine has a two socket mainboard with one shared memory bus. These parallelization demonstrates the effect of an efficient memory bandwidth requirement optimization. The thread level parallelization is based on the two stage blocking method described above. The grid is cut into two halves. Each thread works on one half. The boundary node updates are secured through a semaphore. For a more detailed description of the parallelization we refer to \([Stu06]\).

**Optimizations of the V cycle** An overview about the complete algorithm can be seen in \([10.4]\) The results of the algorithmic optimization in terms of flops and memory accesses is shown in Table \([10.5]\). Table \([10.6]\) shows the distribution among the parts of the complete multigrid algorithm.
Figure 10.3: Standard version of temporal blocking of two iterations

Figure 10.4: Temporal blocking of two iterations with additional blocking planes
Table 10.4: Optimized Multigrid Algorithm for 3D in Pseudo Code

**Optimized V-Cycle in Pseudo-Code**

<table>
<thead>
<tr>
<th>global number</th>
<th>Pre with Pre greater 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>global number</td>
<td>Post with Post greater 0</td>
</tr>
</tbody>
</table>

```plaintext
function Vcycle(half grid Vrot, half grid Vblack, half grid RSred, half grid RSblack) {
  if (Number_Unknowns(V) == 1) {
    Smooth(Vred,Vblack,RSred,RSblack,1)
  } else {
    new coarse half grid Ered
    new coarse half grid Eblack
    new coarse half grid Rred
    new coarse half grid Rblack
    Smooth&Residium(Vred,Vblack,RSred,RSblack,Pre)
    Set_Initial_solution(Ered,Eblack)
    Restriction(Vred,Rred,Rblack)
    Vcycle(Ered,Eblack,Rred,Rblack)
    Interpolate&Correction(Ered,Eblack,Vblack)
    if (finest grid level) {
      Smooth&Residual Norm(Vred,Vblack,Rred,Rblack,Post)
    } else {
      Smooth(Vred,Vblack,RSred,Rblack,Post)
    }
  }
}
```
Table 10.5: Estimate of flops and memory accesses per unknown on one grid level with optimization

<table>
<thead>
<tr>
<th>Component</th>
<th>Flops</th>
<th>Memory throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Itanium</td>
<td>Read</td>
</tr>
<tr>
<td>Smoother &amp; Residual</td>
<td>$7 \cdot i + \frac{8}{2}^2$</td>
<td>2</td>
</tr>
<tr>
<td>Restriction</td>
<td>$\frac{11}{\sqrt{2}}$</td>
<td>$\frac{1}{2} + \frac{1}{3}$</td>
</tr>
<tr>
<td>Interpolation &amp; Correction</td>
<td>$\frac{5}{\sqrt{2}}$</td>
<td>$\frac{1}{2} + \frac{1}{5}$</td>
</tr>
<tr>
<td>Smoother &amp; Residual Norm</td>
<td>$7 \cdot j + \frac{10}{2}^2$</td>
<td>2</td>
</tr>
<tr>
<td>Smoother</td>
<td>$7 \cdot j^2$</td>
<td>2</td>
</tr>
</tbody>
</table>

1 Calculation of residual norm only for finest grid level
2 i Presmoothing- and j Postsmoothing-Iterations

Table 10.6: Distribution on the components for a optimized V(2,2)-cycle

<table>
<thead>
<tr>
<th>Component</th>
<th>Flops (Itanium)</th>
<th>Memory- accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smoother &amp; Residual</td>
<td>46 %</td>
<td>39 %</td>
</tr>
<tr>
<td>Restriction</td>
<td>3 %</td>
<td>9 %</td>
</tr>
<tr>
<td>Interpolation &amp; Correction</td>
<td>3 %</td>
<td>13 %</td>
</tr>
<tr>
<td>Smoother &amp; Residual Norm</td>
<td>44 %</td>
<td>34 %</td>
</tr>
<tr>
<td>Smoother</td>
<td>5 %</td>
<td>5 %</td>
</tr>
<tr>
<td>Number per Unknown</td>
<td>ca. 43.6</td>
<td>ca. 8.7</td>
</tr>
</tbody>
</table>
10.3 Results

10.3.1 Smoother

Addressing of Multi Dimensional Arrays  The first optimization targets the addressing of the three dimensional fields. Figure 10.5 shows different addressing implementations for a straightforward implementation of the Red-Black Gauss Seidel method in 3D. The smoother requires grids of different sizes in dynamically allocated memory (C, dynamic allocation). In contrast to a static three dimensional array (C, static 3D-Array) the compiler as can be seen in Figure 10.5 is not able to generate efficient code. An analysis of the compiler generated assembler code revealed that while the compiler in all cases was able to generate software pipelined loops with appropriate prefetching, it failed for the dynamic case to construct sensible pipeline stages, which take memory and instruction latencies into account. Also compiler flags and pragmas did not improve this situation. An solution to this problem is to use an one dimensional array and implement the addressing of line and plane through an offset variable. In case that line and plane sizes are known at compile time, this results in a significantly better performance (C, optimized). For unknown line and plane sizes the improvements still are small. It is possible to reach the same performance for dynamic sizes using an direct assembler version, but using the same addressing as in the optimized compiler version. Performance drops in the results occur around $50^3$ unknowns, at this point not all data of two half iterations fits into L3 cache. A second drop occurs at $200^3$ unknown, now not even a plane fits into L3 cache. The size of the L2 cache has no direct influence in these measurements, as long as it still can hold the data of the previously updated line.

Memory Layout  The effects of separating the red and black sub grids develop their full effectiveness with further optimizations steps. The overall memory bandwidth requirements are decreased. On the other hand the addressing gets more complicated with different stencils and the padding and alignment results in an higher memory consumption in the caches resulting in an earlier drop out of cache as can be seen in Figure 10.6.

In Cache Performance  The first aim is to reach best possible loop performance in cache. Base is a Red-Black Gauss-Seidel smoother implemented in assembler. In order to fit all data in cache the grid size is only varied in x direction whereas y and z directions are fixed to 12 (inclusive boundary values) (see Figure 10.7). Against the baseline version overlapping of loops and reuse of rotating register gives a significant performance increase, especially with short loop lengths. The best version is the fully optimized kernel also used later in the blocked versions. Here two values at a time in four neighboring lines are calculated. Additional to rotating register reuse and full loop overlap intermediate results are reused. On the shape of the graphs the vector like characteristics can be seen reaching full performance with long vector lengths.
Figure 10.5: Addressing problems

Figure 10.6: Memory layout optimizations
Memory Results  Based on the optimized smoother two versions of a blocked smoother are presented. The temporal blocking for the benchmarks blocks three iterations. One blocking two complete planes and another allowing two stage blocking with sub blocks. The grid size was chosen in a way, so that the total memory consumption is constant around 2 GB. At the first measurement point the grid size is $33 \times 33 \times 123248$, the last one has a grid size of $513 \times 513 \times 510$ points including boundary points.

The standard blocked variant has a rapidly decreasing performance as soon as the plane size increases (see Figure 10.8). For the case, that the data of the last two planes cannot be held in cache, the performance decreases on the level of the unblocked example. The version with two stage blocking inserts additional cuts of the planes. The larger the planes get the more sub blocks have to be introduced leading to the need of multiple loading of data and overhead of the special cut plane treatment. It can be seen in Figure 10.8 that the performance for the sub block version also slowly decreases. A further significant increase in performance can be achieved by smoothing the overall memory accesses and introducing a anticipating two stage prefetching. The total grid size is the same as in the previous measurement. Figure 10.9 shows, that this version shows better results for all plane sizes. In all previous results there occurred occasional performance drops for certain grid sizes. The line length of a half grid are near the for cache conflicts sensitive line lengths $2^n$. While we did not succeed to find an analytic solution for this problem on the present machine it was possible to experimentally find proper padding values to smooth out those performance drops 10.10.
Figure 10.8: Blocked implementation

Figure 10.9: Extended Prefetching and blocked smoother
10.3.2 Overview: Results for serial smoother

In Figure 10.11 the results for the final optimized smoother are shown. For the grid now cubic sizes are used. Beginning from 3 blocked iteration the performance increase is small, because the performance is already limited by cache bandwidth and arithmetic limits. Still it could shown, that it is possible for the 3D case to reach performance increases similar to the results in 2D.

10.3.3 Parallelization of Smoother

If multiple processors share a single memory bus parallelization is only sensible, if the bus is not already saturated by one processor. Commonly numerical algorithms operating on large data sets do not profit from a parallelization on this type of machine. These results also show how effectively the applied optimizations lower the memory bandwidth requirements. In figure 10.12 it can be seen that the version with only one iteration blocked does not profit from a second processor while with three iterations blocked there is a significant performance increase. An overview of the parallel results with regard to blocked iterations can be seen in figure 10.13

10.3.4 Results for the Optimized Multigrid Method

As soon as the smoother is highly optimized, also the optimization of other components of the overall multigrid algorithm get more promising. In Figure 10.14 two runtime results for a single
Figure 10.11: Overview optimized serial smoother

Figure 10.12: Parallelization of smoother
V-cycle are shown. Compared are different number of pre- and post-smoothing steps and the influence of the fused residual calculations with and without parallelization. The V(2,2)-cycle profited most from fusing residual calculations and smoother. Also in the parallel version the fusing showed the biggest performance increase. The V(3,3)-cycle equally profits from fusing and parallelization while the V(4,4)-cycle profits most from parallelization. This of course is a reason of the lower memory bandwidth requirements for the V(3,3) and V(4,4)-cycle. The largest overall improvement is seen for the V(3,3)-cycle with around 29%.

10.3.5 Analysis of convergence behavior

The initial motivation of applying the multigrid method is to quickly reach a solution with a desired accuracy and not the number or execution speed of the involved instructions. In all above measurements this factor was not taken into account. Figure 10.15 shows the convergence behavior against runtime for the serial version of the optimized multigrid code. To reach the minimal residual norm the variant with a V(2,2)-cycle takes 12 V cycles whereas with the V(4,4)-cycle only 10 V cycles are needed. The V(2,2)-cycle is executed so much faster, that despite the worse convergence rate it reaches the required accuracy faster. Despite the faster execution of a V(3,3)-cycle smoother component overall more instructions are needed which are not compensated by the better convergence behavior and the more efficient execution. Things change if the parallel version is considered. Here the V(3,3)-cycle reaches the required accuracy fastest. The V(3,3)-cycle is the best compromise between convergence rate, efficient execution and overall instructions necessary. The V(2,2)-Cycle does profit least.
Figure 10.14: Runtimes for a single V-Cycle with a $513 \times 513 \times 513$ grid

from parallelization while the V(4,4)-cycle profits not enough.

### 10.4 Validation of IA64 results

One motivation to choose the IA64 architecture for an optimized 3D multigrid implementation were the disappointing efforts for a 3D solver on x86/x86-64 architectures. In [Stu05] an optimized 3D solver for x86/x86-64 very similar to this one was implemented. It also involves a two stage blocking method implemented in assembler and uses explicit prefetching techniques. Figure 10.16 shows a comparison between the optimized serial Itanium 2 code and optimized 3D codes for the Pentium 4 and the Athlon 64 test machines. All test machines have a comparable theoretical arithmetic and memory bandwidth performance (For the exact machine characteristics refer to Appendix B). While the x86 architecture show a large decrease when leaving the cache influenced region the Itanium 2 can sustain a very high performance also in the memory region. Its tight control over prefetching together with the full control of the scheduling in software pipelined loops not possible on the other architectures permits high performance with enhanced anticipating prefetching. The results show that this architecture, if used with highly optimized code, can reach a very high efficiency even for demanding scientific algorithms.
Figure 10.15: Convergence behavior on a $257 \times 257 \times 257$ grid
Figure 10.16: Comparison of optimized Smoothers in 3D on different architectures
Part IV

Conclusions and Future Work
Chapter 11

Conclusions and Future Work

11.1 Conclusions

In [Wei01] it was shown that memory bandwidth problems for iterative scientific algorithms can be overcome on cache based architectures by using cache blocking techniques. Later in [Kow04] these techniques were refined and applied to different algorithms. In that work problems occurred with applying these techniques on recent architectures as the Pentium 4. We have shown in this thesis that it is possible to reach a high efficiency for our class of algorithms on modern architectures. The key to reach high performance in comparison to previous work is to find a compromise between the demands of the hardware prefetcher and the code characteristics of blocking techniques. Moreover the elimination of the influence of the compiler on the final instruction code in order to use available hardware resources in the best possible way was crucial for success. For our class of algorithms the initial memory bandwidth bottleneck can be overcome by applying cache blocking techniques. To reach a high efficiency on recent architectures it is necessary to take also other issues as data prefetching and SIMD execution into account.

Current compilers are not able to fully utilize hardware features as SIMD and the memory hierarchy for our class of algorithms. While it is beyond the scope of this work it is still of major interest how available hardware can be used more efficiently. Previous developments, as [ILP] tried to shield the programmer from hardware specific optimizations by means of enhanced compiler technology or were designed to work in a transparent way as the memory hierarchy. The hope that this is also the case for recent developments was not confirmed. The architectures get more complicated and much of this complexity is exposed to the software. The strategy to further increase hardware performance is to add parallelism on multiple levels: The instruction level with [ILP], the data level with [SIMD] and on thread level with CMP [Gse07]. Compiler technology with established programming languages is not able to keep pace with these developments.

This work concentrated on the question if recent architectures can be used efficiently for scientific computing algorithm. We have shown that modern architectures are suitable for
scientific computing and reach a good efficiency. While for a common programmer it is not suitable to program parts of the code in assembler, possible solutions to improve this situation are found in Section 11.3. In section 11.2 we comment on the relationship of the insights of this work to recent hardware developments. Section 11.4 introduces thoughts for possible directions of further research.

11.2 Chip Multi Processors

The recent generation of processors introduces multiple processors on a single chip as a technique to further increase performance of computers. This can be several identical cores connected by a shared memory hierarchy or hybrid designs with specialized processing units as e.g. the Cell architecture or an integrated graphics co-processor on chip. All these new designs can be grouped under the term Chip Multi processors (CMP). These developments introduce concurrency to the software, which imposes many problems to software development in terms of efficiency and productivity. The results and insights of this thesis are in principle also valid for CMP architectures. Also on CMP architectures arithmetic performance is based on SIMD and a sensible usage of a memory hierarchy. Both hardware optimizations which are difficult to be used efficiently by compilers for complicated algorithms. The problem on the memory hierarchy area are getting more severe, because the memory hierarchies are getting complex with caches shared by multiple cores and the introduction of Non Uniform Memory Architecture (NUMA) memory architectures. Additionally CMP adds all implications and problems related to parallel programming. The problems of low efficiency is still present on CMP systems but complexity is increased and thread level parallelism is added. Recent research concentrates on how to use multiple processing units on a chip in a sensible way and still keep the programmability manageable. Still the single core performance is as important as on a common parallel machine. The fact that the hardware can only be exploited by implementing architecture aware software is even more valid today. For performance critical applications it was always necessary to program architecture aware, what is changing is that the performance penalty between an optimized and an unoptimized code is getting larger. There is a lot of research targeting a sensible and productive use of CMP architectures. Examples for research related to multi core architectures are:

- Cilk [Ran98, Lee05], a language for multi threaded parallel programming based on ANSI C.

- The research group of Keshav Pingali with efforts in automatic parallelism and locality enhancements of programs [BKPP08, KP08, KPR+08].

- The Bebop (Berkley Benchmarking and Optimization Group) [WDC+08] which is part of the ParLab project [ABC+06].
11.3 Related Work

This section gives an short overview (with no claim for completeness) about research groups targeting similar questions as in this thesis or trying to find solutions to bridge the gap between software and recent hardware developments.

The following research groups are researching similar questions as covered in this thesis:

- Atlas (Automatically Tuned Linear Algebra Software) automatic generation of optimized numerical software for modern computer architectures and compilers [WPD01].
- FFTW is a C subroutine library for computing the discrete Fourier transform [FJ98].
- The Spiral research group addresses the automatic generation of highly optimized libraries [PMJ05].
- Research group Advanced Scientific Computing at the Technical University in Vienna. Member of the Aurora SFB [PKLU05].
- Keshav Pingali and Kamen Yotov with work to combine model based with search based optimizations [YPS05].
- Research group of William Jalby working on low level code performance and techniques to improve efficiency on recent architectures [DJ08], [BDC07] and [JLT06].
- Darpa High Productivity Computing Systems program [SB04].

There are several efforts to address above mentioned problems on a programming language level. The new CMP processors need an integrated effort involving the programming language and also the operating system. An area not investigated in this thesis are the opportunities interpreted or byte code languages offer in terms of optimization and runtime code adoption. An example for such an approach is the experimental parallel programming language X10 [CGS05]. Other projects targeting the problem with more traditional approaches are Unified Parallel C (UPC) and Co-Array Fortran (CAF) [CDMC05]. Other approaches involve self adopting code as e.g. the Intel Ct programming language [Int07].

11.4 Future Work

It is necessary to draw consequences from the fact that the traditional software implementation chain is not capable to fully exploit current architectures and think about possible solutions to close the gap between hardware developments and traditional software construction chains. The used to be suitable to high level language compilers. New hardware optimization techniques introduced new concepts in the which are not reflected in the high level languages. The current solution to fully expose the programmer to this new complexity is no satisfying solution. High level languages were introduced to increase productivity and maintainability of software. The situation now is similar to former times. Programmers
spend more time on struggling with language level and processor architecture issues than on concentrating on the algorithms and the design of the software. And still the result is not satisfying in terms of performance, productivity and also reliability. Besides introducing new programming paradigms and programming languages one possible solution to this problem is to question the role of the high level languages as they are common today and introduce a model centric programming approach. This approach can be on an abstract or on a domain specific level. The result must be a direct generation of machine code without the step of high level language code. By that there is the chance that the computer architecture and the language constructs in a model driven language are reflecting each other. Model driven development is already common in the area of object oriented programming. Still many efforts here waste a lot of potential of this approach by still generating high level language code instead of directly connecting to the processor architecture. Especially in the high performance computing area a model driven development could bring huge improvements as well in productivity as also in terms of performance.
Part V

Appendix
There were early concepts for a programmable machine computing 1835 by Charles Babbage. Later it was shown that his theoretical concepts were right and lead to a working computing machine. Before 1940 mechanical and electrical analog computers were considered state of the art. Analog computers use continuously varying amounts of physical quantities, such as voltages or currents, or the rotational speed of shafts, to represent the quantities being processed.

There are a number of key inventions that enabled what we call computer today. Probably the root of all other things to come is the master thesis of Claude Elwood Shannon (1916-2001) titled A Symbolic Analysis of Relay and Switching Circuits from 1938 [Sha38]. Shannon owned two Bachelor degrees, one in electrical engineering and one in mathematics. He proved that it is possible to solve boolean algebra and binary arithmetic problems using arrangements of relays. His work was the foundation for digital circuit design and enabled the basic construction of every computer till today.

The first electrical digital computers came up in the early forties. They used vacuum tubes as relays and punch cards as main storage. It is difficult to name one machine the first electronic computer. Around 1945 several projects build more or less general purpose machines. In Germany Konrad Zuse, a genius computer pioneer, invented many developments before anybody else. He build a electrical mechanical stored program computer in the late thirties already using binary arithmetic, without having any touch with the research communities in the US [Cer81]. Still he was an engineer, who only was interested in building a working machine without realizing the fundamental impact.

One of the first operational general-purpose electrical computing machines was ENIAC used for computing artillery fighting tables and developed at the Moore School of Electrical Engineering at the University of Pennsylvania. The major drawbacks were limited storage and tedious programming. Programming was done by plugging cables and setting switches, while the data was entered on punched cards [Wil06]. 1944 John von Neumann, an Hungarian mathematician, joined the project. The group wanted to improve the way programs were entered and thought about storing programs as numbers. This idea was fundamental for the success of the computer. Von Neumann summarized and clarified these thoughts in a memo proposing
a stored-program computer called Electronic Discrete Variable Computer (EDVAC) [Wil93]. This was the birth of the term von Neumann architecture. The idea of a stored-program computer existed in that group long before von Neumann joined. The term von Neumann architecture therefore overestimates the role of von Neumann. From here on the term stored-program computer is used instead. All relevant modern computers are still based on the stored-program architecture. It describes the basic design of a computer, in which the storage is separated from the processing unit. A key design decision is to use a single storage structure to hold both instructions and data. This implies that the hardware is controlled by instructions, which are equivalent to data. All instructions a given processor understands are summarized as ISA. A computation is then described by a sequence of instructions. Such a computing machine is very flexible and can solve various tasks on the same hardware. Figure A.3 shows the elementary parts of a stored-program computer. It is build of five basic components: input, output, memory, Arithmetic Logic Unit (ALU) and control unit. ALU and control unit are often combined and called processor. Execution in a stored-program computer is strictly sequential. The first operational, full scale stored-program computer was EDSAC build 1949 at the University of Cambridge. Maurice Vincent Wilkes, a British computer scientist, got in touch with the ideas of the EDVAC computer. He decided that this was the right way to build a computing machine and started to develop and build EDSAC [Wil97]. So called first generation stored-program computers used vacuum tubes as switches. One of the first commercial computers in that era was the IBM 701, introduced 1952. A major step and leading to the second generation computers is the introduction of transistors and printed circuits in the hardware design of computers. This new generation of machines was less power consuming, smaller and more reliable. In these early days of computing each machine had its dedicated instruction set. In 1960 IBM enabled another major step to make the software concept a success by creating the term computer architecture. The idea was, that a family of computers of the same architecture should be able to run the same software. The IBM System/360, announced 1964, was the first effort to introduce a common architecture shared by a family of computers [Kee04].

One computer of this generation, which stands out and anticipated many future developments, was the CDC 6600 released 1964 [Tho70]. It was designed by Seymour Cray and can be seen as the first supercomputer. Many of its features where unique at that time. It had co processors to do the logic and IO operations and run the operating system. Its central processing unit had a massively superscalar pipelined design. The ISA was RISC like. It had a dedicated floating point unit and was one of the first computers with a pure transistor based design.

Another boost for the use of computers was enabled by the invention of the integrated circuit. These machines are referred to as third generation computers. Examples for computers from this era are the DEC PDP-8 and DEC PDP-11. This later lead to the invention of the microprocessor or fourth generation computers. The first commercial microprocessor was the Intel 4004 released 1971 [Asp97]. Small and affordable computers were possible now enabling
Figure A.1: ENIAC

Figure A.2: EDSAC

Figure A.3: Von Neumann Architecture
Figure A.4: IBM 360

Figure A.5: CDC 6600
the broad use of computers in all areas of life.

In the early days of stored-program computers hardware was precious. Compiler technology did not exist and programs were implemented directly in machine code. The first computer architects as a consequence tried to reassemble high level languages by providing high-level instructions. These instructions carried out several low-level operations as load from memory, arithmetic operation and a memory store. The other precious good was memory. By using complex instructions and variable instruction lengths the code size was minimized. At that time hardware design seemed to be easier than compiler design, so the complexity went into hardware. The climax of this era of high-level-language computer architecture was the DEC Vax architecture released 1978 [BB90]. In the eighties the development started to change. Improvements in programming languages, compiler technology and memory cost meant, that less programming was done at assembly level, which caused that instruction sets could be measured by how well compilers used them.

Curiously it turned out that compilers did not use the more advanced features of the widespread complex architectures. Only a small subset of instructions was actually used by the compilers for code generation. As the more complex instructions were rarely used they were not implemented efficiently anymore on the processors. The implementation of the complex instruction sets required many transistors in the core logic not available for other resources. New hardware developments as pipelining and superscalar design added a lot of complexity to the chip. A new generation of instruction sets was promoted to enable the implementation of these new developments, the Reduced Instruction Set Computing [RISC] machines. A [RISC] processor typically has fewer transistors dedicated to the core logic. This allowed the designers more flexibility for example to increase the size of the register set and add other features as instruction pipelines. These [RISC][ISA] fitted nicely with the new generation of pipelined processors and eased their hardware implementation. All these new techniques can be grouped under the term Instruction Level Parallelism [ILP]. ILP is up to now the major source for an increase in arithmetic performance.

While the widespread use of [ILP] techniques was introduced as a mainstream technique by the RISC architectures, first hardware implementations and the invention of the core technologies happened much earlier. The first pipelined machine was the IBM Stretch. Dynamic scheduling was first introduces by the CDC 6600 in 1964. It was also the first implementation with several functional units. A milestone for exploiting [ILP] was the IBM 360/91 introduced 1967. Among other things it introduced register renaming, forwarding and dynamic scheduling with Tomasuls algorithm. These techniques are still state of the art today. Compared to those major steps, developments of the last years in basic computer architectures appear in a different light. Up to now only the processor was covered. The other essential part of a stored-program computer is the memory together with the bus connecting the memory to the processor. The first computers as the [ENIAC] used vacuum tubes for storing variables. This was very expensive and not applicable to larger memories. [EDSAC] already used so called mercury delay lines. With this new technology [EDSAC] could already store 512 36-bit words.
Unfortunately these technology was unreliable and still rather expensive. The breakthrough came with the invention of core memory by J. Forrester at MIT in the early fifties. Core memory uses a ferrite core, which can be magnetized, and once magnetized, acts as a store. Core memory was cheaper, faster, more reliable, and had higher densities. It soon became the dominating memory technology and remained so for nearly 20 years. The next step was to use integrated circuits for memory. Also before transistorized memory was used for microcode store and caches, but this was very expensive. The first Dynamic Access Memory (DRAM) was built by Intel 1970. DRAM memory became a comparable improvement in available memory size than core memory.

The stored-program architectures single bus between memory and processor leads to a limitation often called Neumann bottleneck or memory wall. The throughput between memory and processor is rather small in comparison to the memory size. On modern machines even worse the throughput is very small in comparison with the rate the CPU itself can work. If the CPU is required to perform minimal processing on a large amount of data this gives a serious limitation in overall processing speed. As CPU speed and memory size increase much faster than memory throughput the bottleneck has become more and more of a problem. Because memory speed was rather fast compared to processing speed this problem was not realized that much until the late seventies. Still already in the early sixties pioneers foresaw the need for a memory hierarchy to bridge the gap between CPU and memory. 1965 Wilkes published a paper describing the concept of a cache like structure [Wil65]. The first commercial computer with a cache was the IBM 360/85 released 1969. In a paper in the development phase of that machine Gibson at IBM for the first time used the term cache in that context [Lip68]. Multilevel caches are a consequence of the lack of improvement in main memory latency and the higher clock rate of microprocessors. In Table A.1 main parameters of milestone machines are listed.
<table>
<thead>
<tr>
<th>Machine</th>
<th>Year</th>
<th>Registers</th>
<th>Speed</th>
<th>Memory Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENIAC</td>
<td>1944</td>
<td>1</td>
<td>Add 200 ms</td>
<td>20 words</td>
</tr>
<tr>
<td>EDSAC</td>
<td>1949</td>
<td>1</td>
<td></td>
<td>1024 17 bit words (200 ms)</td>
</tr>
<tr>
<td>IBM 701</td>
<td>1952</td>
<td>1</td>
<td>Add 60 ms</td>
<td>2048 36 bit words (30 ms)</td>
</tr>
<tr>
<td>IBM 7030</td>
<td>1961</td>
<td>16 ?</td>
<td>Add 1.5 ms</td>
<td>262144 64 bit words (2 ms)</td>
</tr>
<tr>
<td>IBM 360/85</td>
<td>1964</td>
<td>16</td>
<td></td>
<td>512 kB (1 ms)</td>
</tr>
<tr>
<td>IBM 360/91</td>
<td>1964</td>
<td>16</td>
<td></td>
<td>512 kB (1 ms)</td>
</tr>
<tr>
<td>DEC PDP-8</td>
<td>1965</td>
<td>1</td>
<td>Add 2.6 ms</td>
<td>4048 - 32768 12 bit words (1.5 ms)</td>
</tr>
<tr>
<td>CDC 6600</td>
<td>1963</td>
<td>8</td>
<td>1 MFlops</td>
<td>128000 60 bit words, 0.94 MB)</td>
</tr>
<tr>
<td>DEC PDP-11</td>
<td>1970</td>
<td>8</td>
<td></td>
<td>4000 - 28000 16 bit words</td>
</tr>
<tr>
<td>DEC VAX</td>
<td>1978</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A.1: Overview: Milestone machines
Appendix B

Test platforms

B.1 Pentium 4

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Pentium 4 Prescott 3.2 GHz (31 stage pipeline)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache</td>
<td>Code: 12 k µOP 8-way associative Data: 16 kB 8-way associative, 64 byte cache line length Write-Through, Pseudo-LRU</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1024 kB 8-way associative, 64 byte cache line length Sectored 2 cacheline/sector, Exclusive, Pseudo-LRU</td>
</tr>
<tr>
<td>TLB</td>
<td>64 Entries Fully associative</td>
</tr>
<tr>
<td>Arithmetic Peak</td>
<td>6.4 GFlops</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>6.4 GBytes/s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache Latency</td>
<td>4 cycles</td>
</tr>
<tr>
<td>L1 Cache Read Bandwidth</td>
<td>max 46 GBytes/s</td>
</tr>
<tr>
<td>L1 Cache Write Bandwidth</td>
<td>max 12 GBytes/s</td>
</tr>
<tr>
<td>L2 Cache Latency</td>
<td>min 21 cycles, average 56 cycles</td>
</tr>
<tr>
<td>L2 Cache Read Bandwidth</td>
<td>max 23 GBytes/s</td>
</tr>
<tr>
<td>L2 Cache Write Bandwidth</td>
<td>max 12 GBytes/s</td>
</tr>
<tr>
<td>Memory Read</td>
<td>5810 MByte/s</td>
</tr>
<tr>
<td>Memory Write</td>
<td>4018 MByte/s</td>
</tr>
<tr>
<td>Memory Copy</td>
<td>4651 MByte/s</td>
</tr>
<tr>
<td>Sustained Flops FMA</td>
<td>5653 MFlops</td>
</tr>
</tbody>
</table>
B.2 Athlon 64

<table>
<thead>
<tr>
<th>CPU</th>
<th>AMD Athlon 64 4000+ 2.4 GHz (12 stage pipeline)</th>
</tr>
</thead>
</table>
| L1 Cache | Code: 64 kB 2-way associative, 64 byte cache line length, LRU  
Data: 64 kB 2-way associative, 64 byte cache line length  
Dual-Ported, Banked, Write-Back |
| L2 Cache | 1024 kB 16-way associative, 64 byte cache line length  
Sected 2 cacheline/sector, Exclusive, Pseudo-LRU |
| TLB | 512 Entries 4-way associative |
| Arithmetic Peak | 4.8 GFlops |
| Memory Bandwidth | 6.4 GBytes/s |

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache Latency</td>
<td>3 cycles</td>
</tr>
<tr>
<td>L1 Cache Read Bandwidth</td>
<td>max 31 GBytes/s</td>
</tr>
<tr>
<td>L1 Cache Write Bandwidth</td>
<td>max 31 GBytes/s</td>
</tr>
<tr>
<td>L2 Cache Latency</td>
<td>min 11 cycles, average 13 cycles</td>
</tr>
<tr>
<td>L2 Cache Read Bandwidth</td>
<td>max 11 GBytes/s</td>
</tr>
<tr>
<td>L2 Cache Write Bandwidth</td>
<td>max 11 GBytes/s</td>
</tr>
<tr>
<td>Memory Read</td>
<td>6096 MByte/s</td>
</tr>
<tr>
<td>Memory Write</td>
<td>6058 MByte/s</td>
</tr>
<tr>
<td>Memory Copy</td>
<td>6035 MByte/s</td>
</tr>
<tr>
<td>Sustained Flops FMA</td>
<td>4499 MFlops</td>
</tr>
</tbody>
</table>
B.3 Core 2

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Xeon 5160 3.0 GHz Dual Core (14 stage pipeline)</th>
</tr>
</thead>
</table>
| L1 Cache | Code: 32 kB 8-way associative, 64 byte cache line length, LRU  
          | Data: 32 kB 8-way associative, 64 byte cache line length, Write-Back |
| L2 Cache | 4 MB 16-way associative, 64 byte cache line length  
          | shared, Write-Back |
| TLB | 128 Entries 4-way associative |
| Arithmetic Peak | 12 GFlops |
| Memory Bandwidth | 10.6 GBytes/s |

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache Latency</td>
<td>3 cycles</td>
</tr>
<tr>
<td>L1 Cache Read Bandwidth</td>
<td>max 58 GBytes/s</td>
</tr>
<tr>
<td>L1 Cache Write Bandwidth</td>
<td>max 54 GBytes/s</td>
</tr>
<tr>
<td>L2 Cache Latency</td>
<td>14 cycles</td>
</tr>
<tr>
<td>L2 Cache Read Bandwidth</td>
<td>max 21 GBytes/s</td>
</tr>
<tr>
<td>L2 Cache Write Bandwidth</td>
<td>max 19 GBytes/s</td>
</tr>
<tr>
<td>Memory Read</td>
<td>3313 MByte/s</td>
</tr>
<tr>
<td>Memory Write</td>
<td>5280 MByte/s</td>
</tr>
<tr>
<td>Memory Copy</td>
<td>4029 MByte/s</td>
</tr>
<tr>
<td>Sustained Flops FMA</td>
<td>9224 MFlops</td>
</tr>
</tbody>
</table>

B.4 Itanium2

The Itanium 2 is the second implementation of the IA64 ISA. It is in order execution. The scheduler considers two instruction bundles at once. It can therefore start and also execute 6 instructions per cycle. If bundles are executed new bundles are fetched (Bundle Rotation). The scheduler has to wait with the execution of further bundles if:

- it finds a stop
- there is a jump
- no execution unit is available or the issue port is blocked
- there is a cacheline boundary between two bundles
### APPENDIX B

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Itanium 2 1.4 GHz (Madison)</th>
</tr>
</thead>
</table>
| L1 Cache | Instruction: 16 kB, 32 byte cache line length, 4-way associative, latency 1 cycle  
Data 16 kB, 64 byte cache line length, 4-way associative latency 1 cycle, bypassed for floating point data |
| L2 Cache | 256 kB, 128 byte cache line length, 8-way associative, latency min. 5 cycles |
| L3 Cache | 1.5 MB, 128 byte cache line length, 4-way associative, latency min. 12 cycles |
| TLB | Arithmetic Peak: 5.6 GFlops  
Memory Bandwidth: 6.4 GBytes/s |
Appendix C

Results of the Dime Codes

C.1 Description of Different Dime Versions

For a detailed description we refer to the PHD [Wei01].

**rb1**: Standard implementation with one half iteration per sweep.

**rb2 und rb3**: Full iteration in one sweep of the grid. rb2 directly calculates the black point $U_{x,y-1}$ directly following the calculation of the above located red point $U_{x,y}$. rb3 uses full lines as blocks and calculates after all red point in row $y$ the black points in row $y - 1$.

**rb4, rb5**: Extension of version rb2 and rb3 with $i$ iterations in one sweep. rb4 starts as rb3 but updates the above red and black line respectively according to the number of iterations per sweep. rb5 combines this with rb2. In two consecutive lines directly after updating the red point the above black point gets updated.

**rb6, rb7 und rb8**: rb6 are an extension of rb2. After the update of a red point the black and red points located behind it get updated. rb7 and rb8 are unrolled versions of rb6. rb8 does additional calculation reordering to support the compiler in scheduling.

**rb9**: rb9 implements a skewed blocking blocking technique.
C.2 Results

Figure C.1: Plain unblocked 2D dime versions
Figure C.2: Version rb4
Figure C.3: Version rb5
Figure C.4: Version rb6
Figure C.5: Version rb7
RESULTS OF THE DIME CODES

Figure C.6: Version rb8
Figure C.7: Version rb9
Appendix D

x86 Instruction Set Architecture

The purpose of this section is to give an overview for the relevant parts of the x86 architecture. This overview is limited to the most common features and 32 bit. x86 assembler code can be written in two different syntax forms: AT&T and Intel syntax. While the default syntax used by the GNU assembler is AT&T it also understands Intel syntax. In this work Intel syntax is used exclusively. A complete instruction set reference can be found in the Nasm assembler manual [Web08].

D.1 Register set

An overview about common registers is given in table D.1. Additionally there are status and control registers.

D.2 Instruction set

The x86 architecture has a large number of instructions, this is partly caused by backward compatibility, but also because its CISC character. There are five instruction types always involving up to two operands allowed for arithmetical, logical and data transfer instructions as illustrated in table D.2. Opcode lengths vary from 1 byte to 17 bytes with 0 to 2 operands. In contrast to RISC architectures nearly every instruction can have a memory operand. There are 4 basic memory addressing modes listed in table D.3. A reduced overview of for this thesis relevant x86 instructions can be found in table D.4.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA Op1, Op2</td>
<td>load effective address</td>
<td>Calculates offset of second operand and stores it in first. First operand has to be a register, second a memory reference</td>
</tr>
</tbody>
</table>
### APPENDIX D

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV Op1, Op2</td>
<td>move</td>
<td>Copies operand 2 in operand 1. Operand 2 can be a constant, memory reference or register. Operand 1 can be a memory reference or register. At least one operand has to be a register.</td>
</tr>
<tr>
<td>PUSH Op</td>
<td>push</td>
<td>Pushes a constant or register on the stack. Implicitly decrements the stack pointer 4 bytes.</td>
</tr>
<tr>
<td>POP Op</td>
<td>pop</td>
<td>Pops a value from the stack. Operand has to be a register. Implicitly increments the stack pointer 4 bytes.</td>
</tr>
<tr>
<td>RDTCS</td>
<td>read time-stamp counter</td>
<td>Reads out the current value of the time-stamp counter in the EDX:EAX register pair.</td>
</tr>
</tbody>
</table>

#### Jump Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP Op</td>
<td>jump</td>
<td>Unconditional jump. Operand can be a constant address, register or memory reference.</td>
</tr>
<tr>
<td>JA Op</td>
<td>jump if above</td>
<td>Conditional jump. Jump if Carry and Zero Flag are not set. Operand can be a constant address, register or memory reference. Identical to JNBE.</td>
</tr>
<tr>
<td>JAE Op</td>
<td>jump if above or equal</td>
<td>Conditional jump. Jump if Carry is not set. Operand can be a constant address, register or memory reference. Identical to JNB and JNC.</td>
</tr>
<tr>
<td>JB Op</td>
<td>jump if below</td>
<td>Conditional jump. Jump if Carry is set. Operand can be a constant address, register or memory reference. Identical to JNAE and JC.</td>
</tr>
<tr>
<td>JBE Op</td>
<td>jump if below or equal</td>
<td>Conditional jump. Jump if Carry of Zero Flag are set. Operand can be a constant address, register or memory reference. Identical to JNA.</td>
</tr>
</tbody>
</table>

There are 28 more conditional jump instructions not covered here.

#### Arithmetic Instructions

---

144
### ADD Op1, Op2
- **add**
- Add both operands and stores the result in operand 1. The first operand can be a register or memory reference. The second operand can additionally also be a constant.

### DIV Op
- **division**
- Unsigned division. Operand can be a register or memory reference. Divides the content of `EDX:EAX` by the operand. Stores result to `EAX` and the rest to `EDX`.

### MUL Op
- **multiply**
- Unsigned multiplication. Operand can be register or memory reference. Multiplies operand with `EAX` and stores result into `EDX:EAX`.

### SUB Op1, Op2
- **subtract**
- Subtracts operand 2 from operand 1 and stores the result in operand 1. Operand 1 can be a register or memory reference. Operand 2 additionally can be a constant.

### Comparisons

<table>
<thead>
<tr>
<th>CMP Op1, Op2</th>
<th>compare</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compares first with second operand and sets the flags according result. Internally the first operand is subtracted from second operand. Operand 1 can be a memory reference or register. Operand 2 additionally can be a constant. At least one operand has to be a register.</td>
<td></td>
</tr>
</tbody>
</table>

### Logical Operations

<table>
<thead>
<tr>
<th>AND Op1, Op2</th>
<th>logical and</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performs a bitwise logical and on operand 1 and operand 2 and stores the result in operand 1. Operand 1 can be a register or memory reference. Operand 2 can additionally be a constant. Only one memory reference is allowed.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR Op1, Op2</th>
<th>logical or</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performs a bitwise logical or on operand 1 and operand 2 and stores the result in operand 1. Operand 1 can be a register or memory reference. Operand 2 can additionally be a constant. Only one memory reference is allowed.</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX D

XOR Op1, Op2  | logical xor  | Performs a bitwise logical xor on operand 1 and operand 2 and stores the result in operand 1. Operand 1 can be a register or memory reference. Operand 2 can additionally be a constant. Only one memory reference is allowed. 

NOT Op        | not          | Inverts operand bitwise. 

### Procedure Instructions

CALL Op       | call procedure | Calls a sub procedure addressed with the operand, which can be a register or memory reference. Puts the address of the next instruction on the stack as return address. 

RET           | return from procedure | Jumps to the instruction behind a call which is supposed to be on the stack. 

NOP           | null operation | No effect on the program execution. Only changes the instruction pointer. 

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVAPS Op1, Op2</td>
<td>move aligned packed single</td>
<td>Copies four floating point values from Operand 2 to Operand 1. Operands can be either a 128 bit register or memory reference. One must be a register. Memory references must be 16 byte aligned.</td>
</tr>
</tbody>
</table>

### SIMD Extensions

This is a instruction set extension intended to increase performance for graphics, image processing, speech recognition and video or multimedia application. Main new feature is SIMD execution model (Single Instruction Multiple Data) supported by 128 bit wide registers which can hold 2 double precision floating point values (SSE2) or 4 single precision floating point values (SSE). There are specialized packed instruction, which execute multiple operations with one instruction. This was introduced for single performance and later extended in SSE2 for double precision and quadword integers. SSE was introduced with the Pentium 3, SSE2 with the Pentium 4.
### MOVNTPS Op1, Op2
Move packed single using non-temporal hint

Copies four floating point values from Operand 2 to Operand 1. Operand 1 must be a 128 bit memory reference. Operand 2 a 128 bit register. Memory references must be 16 byte aligned. Provides hint that store is non temporal (needs not be cached)

### Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPS Op1, Op2</td>
<td>Add packed single</td>
<td>Adds four single precision floating point values in Operand 1 and Operand 2 and stores the result in Operand 1. Operand 1 must be a 128 bit register. Operand 2 can also be a 128 bit memory reference.</td>
</tr>
<tr>
<td>DIVPS Op1, Op2</td>
<td>Divide packed single</td>
<td>Divides four single precision floating point values in Operand 1 by Operand 2 and stores the result in Operand 1. Operand 1 must be a 128 bit register. Operand 2 can also be a 128 bit memory reference.</td>
</tr>
<tr>
<td>MULPS Op1, Op2</td>
<td>Multiply packed single</td>
<td>Multiplies four single precision floating point values in Operand 1 and Operand 2 and stores the result in Operand 1. Operand 1 must be a 128 bit register. Operand 2 can also be a 128 bit memory reference.</td>
</tr>
<tr>
<td>SUBPS Op1, Op2</td>
<td>Subtract packed single</td>
<td>Subtracts four single precision floating point values in Operand 1 with Operand 2 and stores the result in Operand 1. Operand 1 must be a 128 bit register. Operand 2 can also be a 128 bit memory reference.</td>
</tr>
</tbody>
</table>

### Cache Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFETCHT0 Op</td>
<td>Prefetch with T0 hint</td>
<td>Prefetch cache line containing byte memory reference in cache. Operand is a memory reference. Prefetch into all cache levels.</td>
</tr>
<tr>
<td>PREFETCHT1 Op</td>
<td>Prefetch with T1 hint</td>
<td>Prefetch cache line containing byte memory reference in cache. Operand is a memory reference. Prefetch into level 2 cache or higher.</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>PREFETCHT2 Op</td>
<td>prefetch with T2 hint</td>
<td>Prefetch cache line containing byte memory reference in cache. Operand is a memory reference. Prefetch into level 2 cache or higher.</td>
</tr>
<tr>
<td>PREFETCHNTA Op</td>
<td>prefetch with Non Temporal hint</td>
<td>Prefetch Data at address in cache. Operand is a memory reference. Prefetch data into non-temporal cache structure.</td>
</tr>
<tr>
<td>SFENCE</td>
<td>store fence</td>
<td>Execute all pending store instructions, before executing any store instructions following the fence instruction.</td>
</tr>
</tbody>
</table>

**Instruction Set Extension: SSE2 (144 instructions)**

### Register Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVAPD Op1, Op2</td>
<td>move aligned packed double</td>
</tr>
<tr>
<td>MOVDQA Op1, Op2</td>
<td>move aligned double quadword</td>
</tr>
<tr>
<td>MOVNTDQ Op1, Op2</td>
<td>store double quadword using non-temporal hint</td>
</tr>
<tr>
<td>MOVNTPD Op1, Op2</td>
<td>store packed double using non-temporal hint</td>
</tr>
</tbody>
</table>

### Arithmetic Instructions
### Table D.5: Instruction list SSE

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPD Op1, Op2</td>
<td>add packed double</td>
<td>Adds two double precision floating point values in Operand 1 and Operand 2 and stores the result in Operand 1. Operand 1 must be a 128 bit register. Operand 2 can also be a 128 bit memory reference.</td>
</tr>
<tr>
<td>DIVPD Op1, Op2</td>
<td>divide packed double</td>
<td>Divides two double precision floating point values in Operand 1 by Operand 2 and stores the result in Operand 1. Operand 1 must be a 128 bit register. Operand 2 can also be a 128 bit memory reference.</td>
</tr>
<tr>
<td>MULPD Op1, Op2</td>
<td>multiply packed double</td>
<td>Multiplies two double precision floating point values in Operand 1 and Operand 2 and stores the result in Operand 1. Operand 1 must be a 128 bit register. Operand 2 can also be a 128 bit memory reference.</td>
</tr>
<tr>
<td>SUBPD Op1, Op2</td>
<td>subtract packed double</td>
<td>Subtracts two double precision floating point values in Operand 1 with Operand 2 and stores the result in Operand 1. Operand 1 must be a 128 bit register. Operand 2 can also be a 128 bit memory reference.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLFLUSH Op1</td>
<td>flush cache line</td>
<td>Invalidate cache line containing byte at address of Operand 1. For pending stores cache line is written back before invalidate.</td>
</tr>
<tr>
<td>LFENCE</td>
<td>load fence</td>
<td>Execute all pending load instructions, before executing any load instructions following the fence instruction. Has no effect on prefetch hint instructions.</td>
</tr>
<tr>
<td>MFENCE</td>
<td>memory fence</td>
<td>Combination of store fence and load fence.</td>
</tr>
</tbody>
</table>
### General Purpose Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>accumulator</td>
<td>GPR0</td>
</tr>
<tr>
<td>EBX</td>
<td>base register</td>
<td>GPR1</td>
</tr>
<tr>
<td>ECX</td>
<td>counter register</td>
<td>GPR2</td>
</tr>
<tr>
<td>EDX</td>
<td>data register</td>
<td>GPR3</td>
</tr>
<tr>
<td>ESI</td>
<td>source index</td>
<td>GPR4</td>
</tr>
<tr>
<td>EDI</td>
<td>destination index</td>
<td>GPR5</td>
</tr>
<tr>
<td>ESP</td>
<td>stack pointer</td>
<td>GPR6</td>
</tr>
<tr>
<td>EBP</td>
<td>base pointer</td>
<td>GPR7</td>
</tr>
<tr>
<td>EIP</td>
<td>instruction pointer</td>
<td>IP</td>
</tr>
</tbody>
</table>

### Other registers

<table>
<thead>
<tr>
<th>Naming</th>
<th>Bits</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST0-ST7</td>
<td>80</td>
<td>register stack for FPU</td>
</tr>
<tr>
<td>MM0-MM7</td>
<td>64</td>
<td>map on the FPU registers</td>
</tr>
<tr>
<td>XMM0-XMM7</td>
<td>128</td>
<td>can hold 4 floats or 2 doubles (SSE2)</td>
</tr>
</tbody>
</table>

Table D.1: Registers on x86

<table>
<thead>
<tr>
<th>Source/destination type</th>
<th>Second source operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Register</td>
</tr>
<tr>
<td>Register</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>Memory</td>
</tr>
<tr>
<td>Memory</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

Table D.2: Instruction types

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register indirect</td>
<td>Address is in register</td>
<td>[EBX]</td>
</tr>
<tr>
<td>Base with displacement</td>
<td>Address = base register + displacement</td>
<td>[EBX+16]</td>
</tr>
<tr>
<td>Base plus scaled index</td>
<td>Address = base register + scale × index</td>
<td>[EBX+ESI*8]</td>
</tr>
<tr>
<td>Base plus scaled index with displacement</td>
<td>Address = base + scale × index + displacement</td>
<td>[EBX+ESI*8+16]</td>
</tr>
</tbody>
</table>

Table D.3: Addressing modes
D.3 Application Binary Interface

The ABI specifies how binary code inter operates. This covers how functions are called, how the stack is organized and the interfaces to the operating system. For writing assembler code besides the data representation, the stack organization and the function call sequence is most important. For the complete x86 specific ABI refer to [AT 97]. The following introduction concentrates on the function call sequence.

Each function has a frame on the runtime stack. The stack grows towards low addresses and is word aligned (4 byte). As can be seen in figure D.1 function arguments are pushed on the stack by the caller. The CALL instruction places the return address on the stack. The base pointer register from the caller is preserved next. Besides register EBP the values of registers EBX, ESI and EDI must be preserved for the caller. Also part of the stack frame is local data allocated on the stack, which must be freed before the RET instruction is called. Integral or pointer return values are passed in register EAX. Floating point return values are returned on top of the floating point register stack.
Figure D.1: Stack Layout
Appendix E

x86-64 Instruction Set Architecture

X86-64 is a superset of the x86 architecture introduced by AMD. Other terms used for this architecture are AMD64 and EM64T and Intel 64. It was announced in 2000 and first released with the AMD Opteron processor in April 2003. It supports besides the 64 bit mode a fully x86 compatible legacy 32 bit mode. X86-64 also changes the ABI. Main features are:

- 64 bit Integers.
- 8 additional general purpose registers.
  New Naming RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8 - R15.
- 8 additional SSE xmm registers (xmm0-xmm15).
- Extended virtual and physical address space.
- SSE instructions are added to instruction core.
- Deprecated x86 instructions are removed in 64 bit mode.
- No Execute bit for improved security.

Application Binary Interface

The complete ABI can be found in [htt07]. This short introduction concentrates on the function call sequence and differences to x86.

The stack is similar to x86, with the difference that all data is 16 byte aligned on the stack. Registers EBP, EBX and R12 to R15 have to be preserved for the caller. Different to x86 parts of the arguments are passed in registers. Because the rules for argument passing are rather complicated for a complete description have a look in [htt07]. In short the ABI introduces memory classes. The INTEGRAL class holds arguments which can be stored in a general purpose register. Floating point types belong to the class SSE, which can be stored in an 128 bit register. Fallback is always the so called MEMORY class, passed in memory via the stack. INTEGER arguments are passed in the next available register in the sequence RDI, RSI, RDX,
RCX, R8, R9. If all registers are assigned further arguments are passed in memory on the stack. SSE class arguments are passed in the next available XMM register. Return values are commonly passed in the RAX register for integral types or in register XMM0 for SSE types.
Appendix F

Introduction to gas assembler

In the following the assembler code for a vector triad in assembler is described step by step. Consider a vector triad implemented in C as follows:

```
Listing F.1: Vector triad implemented in C

triad()
{
    int i;
    for (i = 0; i < SIZE; i++){
        A[i] = C[i] + alpha * B[i];
    }
    return 0;
}
```

The vectors A, B and C are assumed to be declared globally. A implementation in x86 gas assembler using SSE floating point instructions is as follows:

```
Listing F.2: Vector triad implemented in gas x86 assembler

.intel_syntax noprefix
.data .align 16 ALPHA: .double 3.6, 3.6
.text .globl triad .type triad, @function
triad: push ebp mov ebp, esp push ebx push ecx push esi push edi
    movapd xmm0, [ALPHA]
    mov edi, A mov edx, C mov esi, B xor eax, eax
    .align 16 1:
    movapd xmm2, [edx + eax*8]
    movapd xmm1, [esi + eax*8]
    mulpd xmm2, xmm0
```

155
All words starting with a dot are compiler directives and not part of the actual instruction set. The directive `.intel_syntax` instructs the assembler that Intel syntax is used. Object files are divided into segments, the actual name and number of segments is object format dependent. Still all object formats share a minimum set of segments, among those are the `.data` segment for initialized data, the `.bss` segment for uninitialized data and the `.text` segment for instruction code. The directive `.data` instructs the assembler that everything that follows has to be put in the data segment. The `.align` directive tells the assembler to align the address following it to a certain number, in our case 16. This can be seen as a macro: For data the assembler fills bytes until it reaches the aligned address, for instruction code it inserts NOP (no operation instruction). It follows the symbol name and again a compiler directive `.double`, which tells the assembler to generate a binary representation for double values, two times 3.6 in this example. Why two times? The reason is it is intended to use SSE packed instructions. These instructions use 128 bit wide registers as operands holding two double values. Every instruction performs two operations, one for each of the double values. To load values from memory into register also two double values can loaded in one instruction. Therefore it is necessary to generate two double values for our global variable alpha to be loaded with a packed load instruction.

In the next line a symbol is declared with the name triad, the `.globl` directive instructs the assembler that this symbol is visible to the linker in the generated object code. As the ELF object format distinguishes symbols to be function symbols or object symbols it has to be specified that this symbol is a function. Symbol in this context means placeholder for an address. During linkage every reference to this placeholder is than replaced by the final address in the executable. With the label triad the assembler is instructed that the declared symbol of the same name starts here. What follows now is related to the ABI for x86. The ABI specifies agreements and rules how e.g. functions are called and function arguments are passed. This is important to ensure that code compiled by different compilers is binary compatible, which means can be linked together and still work. The ABI is architecture and operating system specific, e.g. x86-64 has a different ABI than x86. The x86 architecture has eight named general purpose registers. For a language as C a memory stack is used to pass arguments and allocate local variables. Despite this choice is in principal independent of the ISA, x86
has special instruction for stack access. While for the management of the stack it is sufficient to have one register holding the address of the current stack end address it is useful to have a second register. The first register is called stack pointer and changes its value every time something is pushed or popped from the stack. This register can also be explicitly changed by the code. As already mentioned on x86 arguments to functions and local variables are passed on the stack, which means they have to be accessed during execution of the function. As the stack pointer moves all the time it is difficult to keep track where e.g. the arguments are located relative to the stack pointer. Therefore a second register, the base pointer is used.

At function entry the base pointer is assigned the value of the stack pointer. Its value keeps constant throughout the function making it easy to access arguments located above and local variables located beneath the base pointer address. The offsets to the base pointer keeps constant. On function exit it is possible to free allocated memory by simply assigning the address of the base pointer to the stack pointer. The value of the EBP register has to be saved on the stack. This is necessary because the function responsible that the register value after the function call is the same value than before. This is specified in the ABI. Then the EBP register is assigned the value of the ESP register, the stack pointer. Beneath that a couple of registers are saved on the stack. In Figure D.1 the layout of the stack is shown. As can be seen the stack grows towards low addresses. Beneath possible function arguments there is the return address on the stack. After a function is called the address of the instruction after the call instruction is pushed on the stack by the caller before the jump to the function entry is executed. The return instruction of the called function after execution pops this return address from the stack and jumps to the address it holds.

Next the variable ALPHA is loaded into the xmm0 floating point register. There is a large number of load instructions available in x86. One task of the programmer (and the compiler) is to choose which one is appropriate in a specific situation. Here the movapd instruction is chosen, which stands for move aligned packed double. This instruction is a packed instruction, which means it acts on both double values held in the xmm register. It performs a 16 byte load and expects the load address to be 16 byte aligned. In this example it is assumed, that there are labels for the arrays. This can be e.g. data declared in a .bss section. To use the arrays it is necessary to load their addresses into registers. The x86 ISA does not distinguish between load and store instructions. There are only generic move instructions. The three addresses of the arrays are loaded in the registers edi, edx and esi. Now the loop follows. The address for the loop entry is 16 byte aligned. For a loop at least one counter register is needed, in this example the general purpose register eax. By performing an xor operation on it it set to zero. Next is the loop label 1, the instructions between the loop label and the jump to the label is the loop body. SIMD instructions are used for the vector triad in this example. The code is RISC like, separating load/store from arithmetic instructions. The x86 ISA also allows memory operands for arithmetic instructions. To dereference a pointer the [] operator is used. It tells the processor to fetch data from the address calculated inside the brackets. The syntax for addressing is \([BASE + INDEX * SCALE + DISP]\). BASE is the register
holding the base address. INDEX is the register holding the counter, whereas SCALE specifies what the data size in bytes (can be one of 1, 2, 4 and 8). DISP is an additional offset e.g. when unrolling a loop. After loading two values of each vector into xmm2 and xmm1, xmm2 and xmm0 are multiplied. The result is stored in the first operand register xmm2. After that the values in xmm1 and xmm2 are added. Every arithmetic instruction stands for two operations and acts on two values simultaneously. While the ISA allows to execute these operations in one cycle the Pentium 4 and Athlon 64 split up these instructions and execute them in two steps. These processors can perform one multiply/add in one cycle. If there are just add operations it is limited to one operation per cycle. The first hardware implementation of the SSE instruction set extension able to perform one SSE packed instruction in one cycle is the Intel Core 2 architecture. The next step is to store the result back. For that the special instruction movntpd is used. This is a SSE2 instruction, too. It stands for move non temporal packed double. Packed again means that it performs a 16 byte move. Non Temporal is a hint. This means the hardware implementation can use the information, but is not forced to do so. With non temporal the implementation is informed, that this move has no temporal locality or more clearly that it is not planned to use this data again in the future. With this information the processor can write the data directly to memory, bypassing the caches. With the non temporal hint it is not necessary to enforce data consistency for this data. The processor therefore can collect subsequent writes in write combining buffers and burst e.g. 64 byte at once to memory. Especially on the Intel Pentium 4 processor it is crucial to use this instruction to get a reasonable memory store bandwidth. After adding two to the counter register the value is compare to the stopping criterion, in this case a constant value. Constant values is hard coded into the instruction and called immidiate operand. The compare instruction as nearly all instructions influences the status flag register. Depending if the carry flag is set or not he jump back to the loop entry is carried out or not. After ending the loop all saved registers are restored. The stack pointer is set to the value of the base pointer. At last the base pointer is restored and the function returns to the caller, by popping the return address from the stack and jumping to this address.
Motivation of the test problem: Heat Equation

For the following relationships a basic knowledge of thermodynamics is needed. Textbooks on Thermodynamics are e.g. [Bec85, BSL60, Hah93]. The reason why these relationships are difficult to understand is that they are based on experience rather than on analysis. Among others there are two central terms in thermodynamics: energy and temperature. These terms are used by habit in daily life, still to give a clear definition of them is not trivial. What energy really is is not known. It can only be described how it behaves and how it can be converted in different forms. From mechanics the terms kinetic and potential energy are known. In order to change these energy forms experience shows, that it is necessary to apply or conserve work. The first imagination of what the abstract term energy is can be written as:

| Energy is a measure for the ability to accomplish work. |

There is a second experience connected to the term energy. If bodies are rubbed (friction work is applied) they heat up.

| Energy is a measure for the ability to heat up something. |

The most important property of energy is that it is impossible to destroy it. It holds the principal of energy conservation.

| In a closed system total energy is not changing. |

The SI unit for energy is Joule, which is a derived unit defined as the energy required to exert a force of one Newton over a distance of one meter. Joule is therefore equivalent to $N \cdot m$ or in base units \( \frac{kg \cdot m^2}{s^2} \).

The term temperature is equally difficult to define. Temperature is a base unit as is length or mass. Therefore there exists a legal agreement which says:

| Base unit of thermodynamic temperature is Kelvin. 1 Kelvin is the 273.16th part of the thermodynamic triple point of water. |

The triple point of a pure substance is the state as a function of \((p,T)\), in which all three phases
(solid, liquid and gas) coexist in thermodynamic equilibrium. Still that does not give any intuition what is the character of temperature. To give a more physical definition of temperature it is necessary to introduce another term: Thermodynamic equilibrium. A system is described by state variables. These are e.g. pressure, temperature, volume, mass and so on. A system is in thermodynamic equilibrium if the following statement holds:

A system is in a state of thermodynamic equilibrium, if its state variables do not change without influence from outside.

The basic physical principal applied is energy conservation. The formula reads as:

\[ \frac{dU}{dt} = -\int_S \dot{q} dS + \int_V \dot{q}_V dV \]  

(G.1)

It states, that the change in internal energy of a fixed volume over time is composed of internal energy sources (e.g. electrical energy or absorption through radiation) and the heat flux over its surface. The minus in front of the heat flux term is necessary, because the normal vector points outside of the volume.

Applying the divergence theorem the surface integral can be converted into a volume integral. Using the relationship \( U = \int_V \rho u dV \) \((V = \text{const.}, \ u \) is the specific internal energy \([\frac{J}{kg}]\)), equation (G.1) can be rewritten in differential form as follows:

\[ \frac{\partial}{\partial t} (\rho u) = -\text{div} \dot{q} + \dot{q}_V \]  

(G.2)

It is assume that there are no internal energy sources, the last term therefore drops out. Equation (G.2) is a consequence of the first law of thermodynamics. Experience shows that a change of internal energy in a closed system is coupled to a change in temperature. The relationship between the addition of heat and the change in temperature can be written as:

\[ \delta Q = dU = C \cdot dT = m \cdot c \cdot dT \]  

(G.3)

The proportional coefficient \( C \) is named heat capacity or specific heat capacity \( c \) if it is based on mass. Using this relationship together with above assumption we get:

\[ \rho c_p \frac{\partial T}{\partial t} = -\text{div} \dot{q} \]  

(G.4)

Heat conduction is described by Fourier’s law. It states, that heat flux is proportional to the temperature gradient. For isotropic medium Fourier’s law states:

\[ \dot{q} = -\lambda \cdot \text{grad}T \]  

(G.5)

The minus says that heat flux occurs from high to low temperatures, meaning is positive down the temperature slope. With the assumption, that the thermal conductivity \( \lambda \) is constant, equation (G.4) transforms to the unsteady temperature equation for solid bodies:

\[ \rho c_p \frac{\partial T}{\partial t} = \frac{\lambda}{\rho c_p} \Delta T \]  

(G.6)
The term \( \frac{\alpha}{\rho c_p} \) is also called thermal diffusivity \( \alpha \). For the steady case the time derivative vanishes and equation (G.6) becomes:

\[
\Delta T = 0 \tag{G.7}
\]

Equation (G.7) is known as Laplace’s equation or if right hand side is not zero (meaning there are internal heat sources) as Poisson’s equation. Because Poisson’s equation is relatively simple, but still of relevance in the field of engineering and natural science it was chosen as test case throughout this work.
Appendix H

German Parts

Zusammenfassung

Für viele numerischen Programme wird der Transport der Daten vom Hauptspeicher in die Register als der wichtigste leistungbegrenzende Einfluss angesehen um hohe Leistung auf derzeitigen Mikroarchitekturen zu erreichen. Diese Tatsache wird oft als memory wall bezeichnet. Umfassende Forschungsprojekte beschäftigen sich mit diesem Punkt auf unterschiedlichen Ebenen. Dies beinhaltet z.B. Programmcodetransformationen und auf die Architektur zugeschnittene Datenstruktur um eine bestmögliche Nutzung der Speicherrhierarchie auf derzeitigen Mikroarchitekturen zu erreichen. Dies vorliegende Arbeit zeigt, das es auf modernen Mikroarchitekturen notwendig ist auch die Anforderungen von SIMD und Datenvorausladetechniken zu beachten um ein hohe Effizienz zu erreichen.

In dieser Dissertation wird die Kette von der Optimierungen auf Hochsprachenebene über den Programmcodegenerierungsprozess und den Beschränkungen und Einflüssen des Instruktions satzes bis zur Mikroarchitektur der zugrundeliegenden Hardware untersucht. Als Ergebnis wird eine Strategie präsentiert um eine hohe Leistung für initial speicherbandbreitenbegrenzte Algorithmen auf modernen Architekturen zu erreichen. Der Erfolg dieser Herangehensweise wird anhand der algorithmischen Klasse der Gitterbasierten numerischen Gleichungslöser: Einer Implementierung eines 2D Rot-Schwarz Gauß-Seidel Glätters für die x86/x86-64 Architektur und eine 3D Mehrgitter Implementierung für die IA64 Architektur.
Appendix I

Curriculum Vitae

General information

Surname, first name: Treibig, Jan
Sex: Male
Date and place of birth: 28th October 1973 in Erlangen, Germany
Marital status: Married
Nationality: German
Current office: Universität Erlangen-Nürnberg
Regionales Rechenzentrum Erlangen (RRZE)
HPC Services
Martensstrasse 1
D-91058 Erlangen
Email: jan.treibig@rrze.uni-erlangen.de
Phone: +49 9131 85-28136
Education and University Studies

Sep. 1980 – July 1982: Primary school in Erlangen
Sep. 1995 – Aug. 1996: Magister Geography/ Modern German History
Sep. 1996 – Feb. 2002: Graduate Engineer in Chemical Engineering

Universität Erlangen-Nürnberg, Germany

Mar. 2002 – Aug. 2006: PhD student at the computer science department at the chair
of Systemsimulation of the Friedrich-Alexander Universität
Erlangen-Nürnberg

Sep. 2006 – Aug. 2008: Software Engineer at Elektrobit Automotive
Since Nov. 2008: Research Assistant at Regionales Rechenzentrum Erlangen
of the Friedrich-Alexander Universität
Erlangen-Nürnberg
Bibliography


APPENDIX I


168


[SGI96] SGI. Optimizing cpu to memory accesses on the sgi visual workstations 320 and 540, 1996.


APPENDIX I


